计算机组成原理

计算机组成原理课程组
（刘旭东、高小鹏、肖利民、牛建伟、栾钟治）

第五讲：指令系统与MIPS汇编

1.1 指令系统概述

❖指令系统的基本问题

- 操作类型：应该提供哪些（多少）操作？
  - 用存、取、加、转移等已经足够编写任何计算程序，但不实用，程序太长。
- 操作对象：如何表示？可以表示多少？
  - 大多数是双值运算（如A←B+C)
  - 存在单值运算（如A←~B)
- 指令格式：如何将这些内容编码成一致的格式？
  - 指令长度、字段、编码等问题

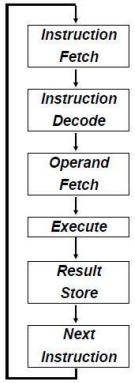
1.1 指令系统概述

❖机器指令的要素

- 操作码(Operation Code)：指明进行的何种操作
- 源操作数地址(Source Operand Reference)：参加操作的操作数的地址，可能有多个。
- 目的操作数地址(Destination Operand Reference)：保存操作结果的地址。
- 下条指令的地址(Next Instruction Reference)：指明下一条要运行的指令的位置，一般指令是按顺序依次执行的，所以绝大多数指令中并不显式的指明下一条指令的地址，也就是说，指令格式中并不包含这部分信息。只有少数指令需要显示指明下一条指令的地址。

1.1 指令系统概述

❖从指令执行周期看指令涉及的内容



Obtain instruction from program storage
指令地址、指令长度（定长/变长）

Determine required actions
指令格式、操作码编码、操作数类型

Locate and obtain operand data
地址码格式、寻址方式、操作数格式和存放方式

Compute result value or status
操作类型、标志或条件码

Deposit results in storage for later use
结果数据位置

Determine successor instruction
下条指令地址（顺序 / 转移）

from： 南大袁春风老师ppt

1.1 指令系统概述

❖ 操作数的位置
- 存储器（存储器地址）
- 寄存器（寄存器地址）
- 输入输出端口（输入输出端口地址）

❖ 操作数的类型
- 数值（无符号、定点、浮点）
- 逻辑型数、字符
- 地址（操作数地址、指令地址）

❖ 操作数的存储方式
- 大端（big-endian）次序：最高有效字节存储在地址最小位置
- 小端（little-endian）次序：最低有效字节存储在地址最小位置

例： Int a; //0x12345678

地址	值
a+0	12
a+1	34
a+2	56
a+3	78

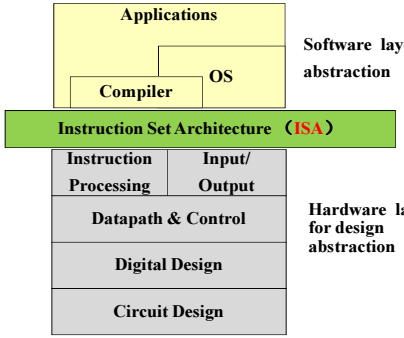
大端次序

地址	值
a+0	78
a+1	56
a+2	34
a+3	12

小端次序

1.1 指令系统概述

❖ 指令集系统结构(ISA)
- 机器语言编程者的视角，机器内部结构和行为能力的指令级抽象



Software layers of abstraction

Hardware layers for design abstraction

1.1 指令系统概述

❖指令集系统架构（ISA）种类
- 大部分ISA都可归类为通用寄存器系统结构
- Register-Memory式ISA（如80X86）
  - 多种指令可以访问内存；
  - 存在寄存器操作数和内存操作数直接运行的指令；
- Load-Store式ISA（如MIPS）
  - 只有装载（LOAD）和存储（STORE）指令可以访问内存
  - 运算指令操作数全部为寄存器操作数；

❖Load-Store是ISA的一种趋势

1.1 指令系统概述

❖指令类型

- 数据传输指令：寄存器与存储器之间，寄存器之间传递数据；
- 算术/逻辑运算指令：寄存器（或存储器）中整型数或逻辑型数据的运算操作。
- 程序控制指令：控制程序执行顺序，条件转移或跳转，子程序调用和返回等；
- 浮点运算指令：处理浮点数的运算。

❖通用寄存器的优势

- 寄存器比存储器快
- 寄存器便于编译器使用
- 寄存器可以保存变量
- 减少存储器访问，提高速度
- 提高代码密度，寄存器地址比存储器地址短

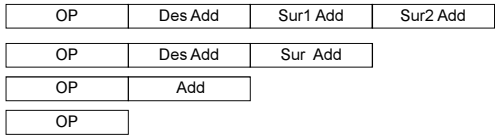
1.2 指令格式

❖指令的表示

- 机器表示：二进制代码形式
- 符号化表示：助记符，如 MOV AX，BX

❖操作数地址的数目

- 三地址： Des ← (Sur1) OP (Sur2)
- 双地址： Des ← (Sur) OP (Des)
- 单地址：累加器作为默认操作数的双操作数型，或单操作数型
- 无地址：隐含操作数型，或无操作数型



1.2 指令格式

❖操作码结构

- 固定长度操作码：操作码长度（占二进制位数）固定不变。
  - 硬件设计简单
  - 指令译码时间开销较小
  - 指令空间效率较低
- 可变长度操作码：操作码长度随指令地址数目的不同而不同。
  - 硬件设计相对复杂
  - 指令译码时间开销较大
  - 指令空间利用率较高

❖指令长度

- 定长指令系统，如MIPS指令
- 变长指令系统：一般为字节的整数倍，如80X86指令

1.3 寻址方式

❖形式地址与有效地址

- 形式地址：指令中直接给出的地址编码。
- 有效地址：根据形式地址和寻址方式计算出来的操作数在内存单元中的地址。
- 寻址方式：根据形式地址计算到操作数的有效地址的方式（算法）

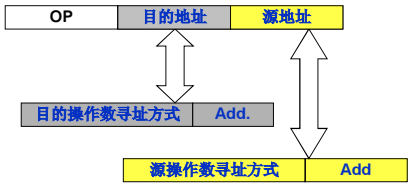
❖常用寻址方式

- 立即寻址
- 寄存器直接寻址
- 寄存器间接寻址
- 基址寻址/变址寻址
- 相对寻址：基址寻址的特例，程序计数器PC作为基址寄存器
- 堆栈寻址

1.3 寻址方式

❖寻址方式的确定

- 在操作码中给定寻址方式：
  - 如MIPS指令，指令中仅有一个主(虚)存地址的，且指令中仅有一二种寻址方式。Load/store型机器指令属于这种情况。
- 专门的寻址方式位
  - 如X86指令，指令中有多个操作数，且寻址方式各不相同，需要各自说明寻址方式。



1.3 寻址方式

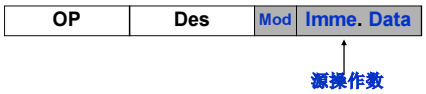
❖指令代码和寻址描述中有关缩写的约定

- OP：操作码
- Des：目的操作数地址
- Sur：源操作数地址
- A或Add： 形式地址（内存地址）
- Mod：寻址方式
- Rn ： 通用寄存器
- Rx ：变址寄存器
- Rb ： 基址寄存器
- SP：堆栈指针（寄存器）
- EA ：有效地址
- Data ：操作数
- Operand ：操作数
- (Rn)：寄存器Rn的内容（值）
- Mem[A]：内存地址为A的单元的内容
- Imme. Data ： 立即数
- XXH：16进制数XX

1.3 寻址方式

❖立即寻址

- 操作数直接在指令代码中给出。



❖说明

- 立即寻址只能作为双操作数指令的源操作数。
- Operand = Imme. Data
- 例：MOV AX,1000H （80X86指令， AX←1000H）
  addi $s1, $s2, 100 （MIPS指令, $s1 ← $s2+100）

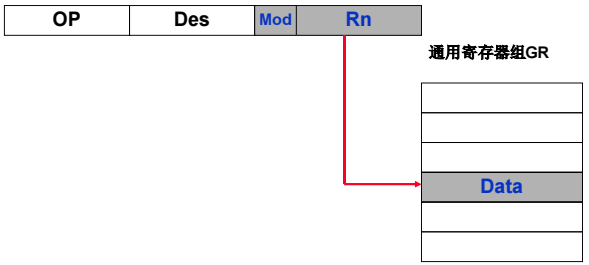
❖思考

- 立即寻址的操作数在什么地方，存储器 or 寄存器？
- 立即数的地址？

1.3 寻址方式

❖寄存器直接寻址

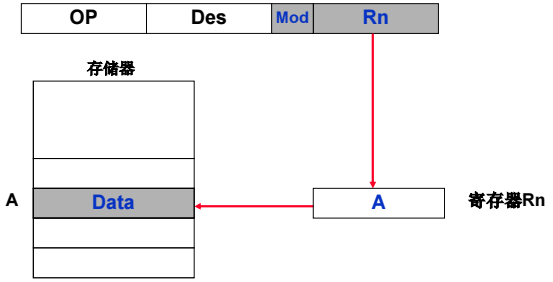
- 操作数在寄存器中，指令地址字段给出寄存器的地址（编码）
- EA = Rn, Operand = (Rn)
- 例：MOV [BX], AX （80X86指令）
  add $s1,$s2,$s3 （MIPS指令， $s1←$s2+$s3）



1.3 寻址方式

❖寄存器间接寻址

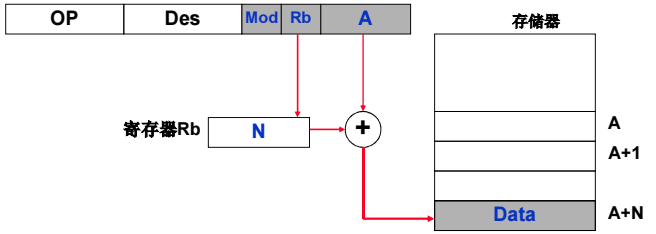
- 操作数在存储器中，指令地址字段中给出的寄存器的内容是操作数在存储器中的地址。
- EA = (Rn), Operand = Mem[(Rn)]
- 例：MOV AX, [BX] （80X86指令, AX ←Mem[(BX)]）



1.3 寻址方式

❖基址寻址

- 操作数在存储器中，指令地址字段给出一基址寄存器和一形式地址，基址寄存器的内容与形式地址之和是操作数的内存地址。
- EA = (Rb)+A, Operand = Mem[(Rb)+A]
- 例：MOV AX, 1000H[BX] （80X86指令，AX←Mem[(Bx)+1000]）
  lw $s1,100($s2) （MIPS指令，$s1 ←Mem[$s2+100]）

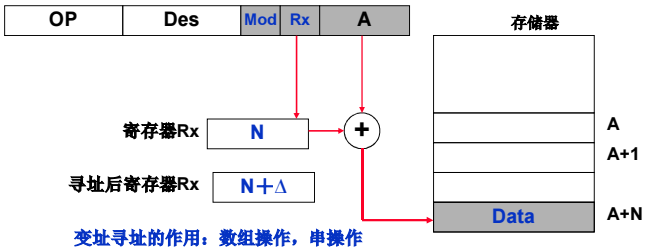


基址寻址的作用：较短的形式地址长度可以实现较大的存储空间的寻址。

1.3 寻址方式

❖变址寻址

- 操作数在存储器中，指令地址字段给出一变址寄存器和一形式地址，变址寄存器的内容与形式地址之和是操作数的内存地址。
- EA = (Rx)+A, Operand = Mem[(Rx)+A]
- 有的系统中，变址寻址完成后，变址寄存器的内容将自动进行调整。
  Rx ← (Rx) + Δ（操作数Data的字节数）
- 例：MOV AX, 1000H[DI] （80X86指令）

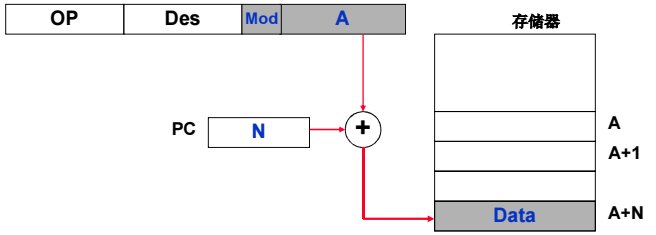


变址寻址的作用：数组操作，串操作

1.3 寻址方式

❖相对寻址

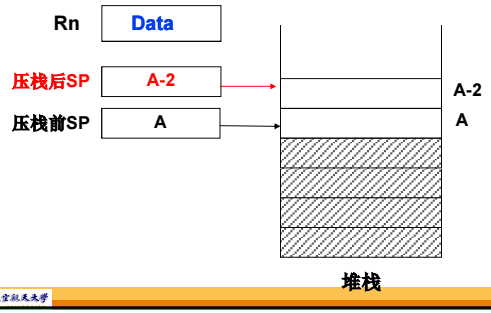
- 基址寻址的特例，由程序计数器PC作为基址寄存器，指令中给出的形式地址作为位移量，二者之和是操作数的内存地址。
- EA = (PC)+A, Operand = Mem[(PC)+A]
- 例：JNE A （80X86指令）
  beq $s1, $s2, 100 (MIPS指令)



1.3 寻址方式

❖堆栈寻址

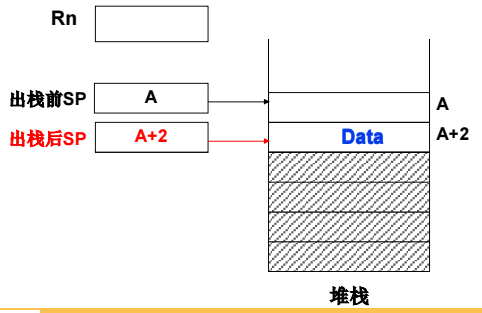
- 堆栈的结构：一段内存区域。
- 堆栈指针(SP)：是一个特殊寄存器部件，指向栈顶
- 压栈操作：PUSH Rn，假定寄存器Rn为16位寄存器
  (SP) ← (Rn)，SP ← (SP)-2



1.3 寻址方式

❖堆栈寻址

- 出栈操作：POP Rn，假定寄存器Rn为16位寄存器
  SP ← (SP) + 2， Rn ← ((SP))



1.3 寻址方式

◆基本寻址方式的对比

指令：OP A，R，.....

假设：A=地址字段值，R=寄存器编号，
EA=有效地址， (X)=地址X中的内容

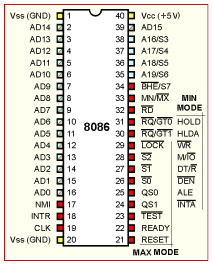
方式	算法	主要优点	主要缺点
立即	操作数=A	指令执行快	操作数幅值有限
存储器直接	EA=A	有效地址计算简单	地址范围有限
寄存器直接	操作数=（R）	指令执行快，指令短	地址范围有限
寄存器间接	EA=（R）	地址范围大	额外存储器访问
相对	EA=A+（R）	灵活	复杂
堆栈	EA=栈顶	指令短	应用有限

第五讲：指令系统与MIPS汇编

2.1 8086/8088指令系统

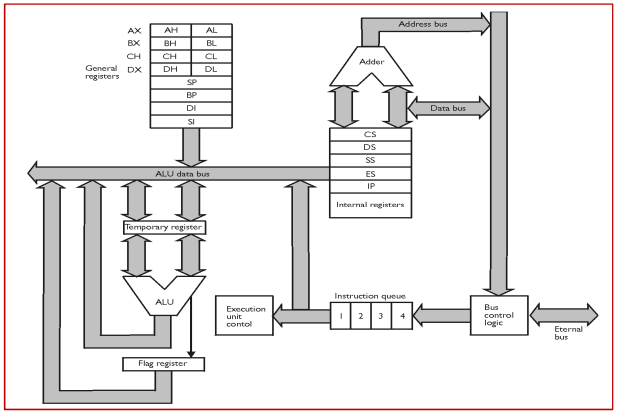
❖8086 / 8088CPU简介

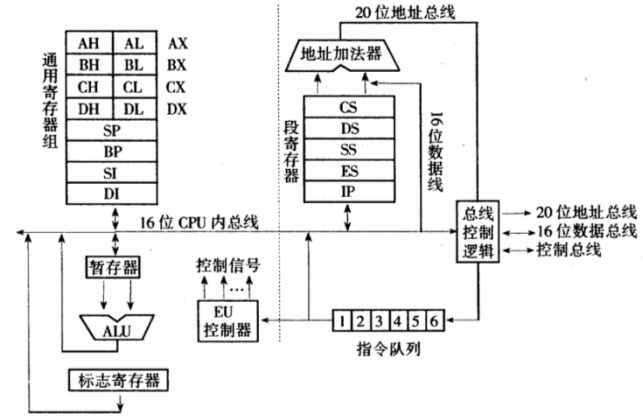
- 1978年由Intel推出，16位微处理器
- CISC（Complex Instruction Set Computer，复杂指令集计算机）处理器
- 29000只晶体管
- 速度可分为5MHz、8MHz、10MHz
- 内部数据总线、外部数据总线均为16位
- 地址总线为20位，可寻址1MB内存
- 1981年，IBM推出首款选用8088 CPU的个人电脑IBM PC



2.1 8086/8088指令系统

❖8086 / 8088CPU结构





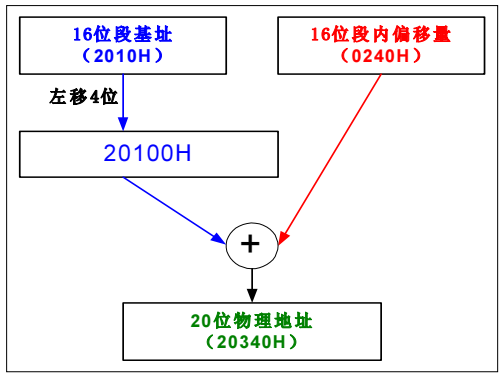
2.1 8086/8088指令系统

❖存储器及其存储器地址结构

- 主存容量为1M（2^{20}），可直接访问的主存物理地址为20位。超过1M的存储空间通过其他方式访问。
- 8086 / 8088机器字长16位，所有寄存器长度为16位，数据总线16位。
- 主存采用分段的结构
- 主存存储单元的地址构成：段基址（16 bits）：段内偏移（16 bits）
- 可执行程序（.EXE）的存储结构：代码段(Code Segment)，数据段(Data Segment)，堆栈段(Stack Segment)，扩展数据段（可选）
- 命令程序（.COM）的存储结构：代码段，数据段和堆栈段必须是同一个段。所以命令程序最大为64KB存储空间。

2.1 8086/8088指令系统

❖存储器地址结构与计算



2.1 8086/8088指令系统

❖存储器单元结构

- 按字节单元编址

❖字节单元
- （2000H）=20H
- （2001H）=10H

❖字单元
- （2000H）=1020H
- （2004H）=5060H

❖双字单元
- （2000H）=30401020H

	高字节	低字节
2000H	10H	20H
2002H	30H	40H
2004H	50H	60H
2006H	70H	80H

存储器

2.1 8086/8088指令系统

❖通用寄存器：数据寄存器(Data Register)

- AX，BX，CX，DX（16位）；
- AH，AL，BH，BL，CH，CL，DH，DL（8位）

注：各寄存器原则上没有固定的应用
- AX：累加器
- BX：基址寄存器
- CX：计数器
- DX：数据寄存器

	8 bits	8 bits
AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL

通用寄存器

2.1 8086/8088指令系统

❖通用寄存器：指针寄存器(Pointer Register)

- 堆栈指针：SP（16 位）
- 基址指针：BP（16位），默认指向堆栈段

❖通用寄存器：变址寄存器(Index Register)

- SI，DI：16位
- 一般情况下，二者使用上无差异，在串操作中，SI 对应源操作数，DI对应目的操作数

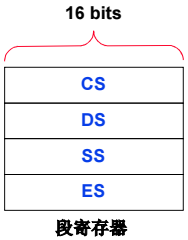
16 bits
SP
BP
SI
DI

通用寄存器

2.1 8086/8088指令系统

❖通用寄存器：段寄存器(Segment Register)

- 代码段(Code Segment)，数据段（Data Segment），堆栈段（Stack Segment），扩展数据段（Extend data Segment）
- 代码段寄存器：CS（16 bits）
- 数据段寄存器：DS （16 bits）
- 堆栈段寄存器：SS （16 bits）
- 扩展段寄存器：ES （16 bits）



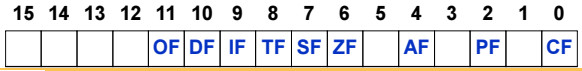
2.1 8086/8088指令系统

❖指令指针IP（Instruction Pointer）

- IP（16 bits)指向代码段中下一条要执行的指令。
- CS:IP 形成下一次要执行的指令的内存地址。

❖标志寄存器FLAGS（Flags Register）

- 16位，记录当前CPU运行程序的各种状态
- 进位标志位 CF（Carry Flag）
- 奇偶标志位 PF（Parity Flag）
- 辅助进位标志位 AF（Auxitiary Flag）
- 零值标志位 ZF（Zero Flag）
- 符号标志位 SF（Sign Flag）
- 溢出标志位 OF（Overflow Flag）
- 单步跟踪标志位 TF（Trace Flag）
- 中断允许标志位 IF（Interrupt-enable Flag）
- 方向标志位 DF（Direction Flag）



2.1 8086/8088指令系统

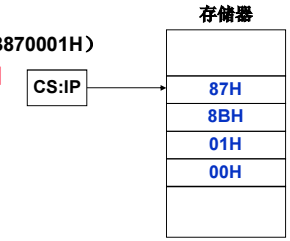
❖寻址方式

- 立即寻址，　　如：MOV AL, 05H
- 寄存器直接寻址，　如：MOV AX, BX
- 基址（变址）寻址，如：MOV AX, 1000H[BX]
- 串操作寻址，　　如：MOVSB //隐含寻址DS: (SI) > ES: (DI)
- I/O寻址，　　如：OUT DX, AL //DX为端口号

基址寻址举例：

MOV AX, 1000H[BX] （指令代码：8B870001H）

- 等价于 MOV AX, DS:[BX＋1000H]
- EA = DS: (BX)+1000H



串操作寻址举例：

- MOV SI, 1000H //DS: (SI)
- MOV DI, 3000H //ES: (DI)
- MOV CX, 0100H
- CLD
- REP MOVSB //SI++; DI++; CX--

2.1 8086/8088指令系统

❖一般双操作数指令格式与编码

- RR型或RS型，必有一个操作数在寄存器中（寄存器直接寻址）
- 长度2～6个字节（前2个字节必须）
- Opcode：操作码（6位）
- d: 方向字段（1位）。在第二个字节中，REG确定一个操作数（寄存器直接寻址），MOD和R/M确定另一个操作数的寻址方式。方向字段d表明REG确定的是源操作数还是目的操作数。
  - d=1, REG确定目的操作数，MOD+R/M确定源操作数
  - d=0, REG确定源操作数，MOD+R/M确定目的操作数
- W：字 / 字节字段（1位）：操作数是字节（8位）还是字（16位）
  - W=1，字（16位）
  - W=0，字节（8位）

2.1 8086/8088指令系统

❖一般双操作数指令格式与编码（续）

- REG：寄存器字段，指明两个操作数中寄存器直接寻址的那个操作数。与W字段配合使用。

寄存器编码表

REG	W=1	W=0
000	AX	AL
001	CX	CL
010	DX	DL
011	BX	BL
100	SP	AH
101	BP	CH
110	SI	DH
111	DI	BH

2.1 8086/8088指令系统

❖一般双操作数指令格式与编码（续）

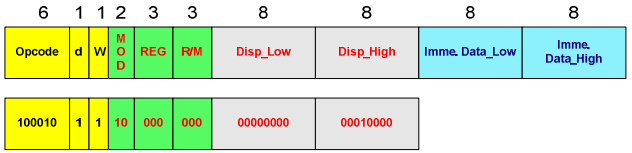
- MOD和R/M：确定另外一个操作数。

R/M	存储器操作数有效地址(EA)			寄存器操作数	
	MOD=00	MOD=01	MOD=10	MOD=11	
				W=1	W=0
000	(BX)+(SI)	(BX)+(SI)+Disp8	(BX)+(SI)+Disp16	AX	AL
001	(BX)+(DI)	(BX)+(DI)+Disp8	(BX)+(DI)+Disp16	CX	CL
010	(BP)+(SI)	(BP)+(SI)+Disp8	(BP)+(SI)+Disp16	DX	DL
011	(BP)+(DI)	(BP)+(DI)+Disp8	(BP)+(DI)+Disp16	BX	BL
100	(SI)	(SI)+Disp8	(SI)+Disp16	SP	AH
101	(DI)	(DI)+Disp8	(DI)+Disp16	BP	CH
110	Disp16	(BP)+Disp8	(BP)+Disp16	SI	DH
111	(BX)	(BX)+Disp8	(BX)+Disp16	DI	BH

2.1 8086/8088指令系统

❖指令编码举例

- MOV AX，1000H[BX][SI]
- MOV的操作码 Opcdoe＝100010
- d=1：目的操作数是寄存器直接寻址
- W=1：16位字操作
- REG=000：表示AX
- MOD=10, R/M=000：另一个操作数EA=(BX)+(SI)+1000H
- 指令前两个字节＝1000101110000000（8B80H）
- 4字节指令编码：8B800010H



2.1 8086/8088指令系统

❖指令类型

- 传送指令：MOV，XCHG，LDS，LEA
- 算术运算指令：ADD，INC，SUB，CMP等
- 逻辑运算指令：AND，OR，NOT，TEST等
- 处理器控制指令：CLC，STC，CLI，STI，CLD，NOP等
- 程序控制指令：CALL，RET，JMP，JNE，INT，IRET等
- 串指令：MOVSB，MOVSW等
- I/O指令：IN，OUT

❖80286新增指令(101)
- 堆栈操作指令
- 有符号数乘法指令
- 移位指令

❖80386新增指令(14)
- 数据传送与填充指令
- 堆栈操作指令
- 取段寄存器指令
- 有符号数乘法指令
- 符号扩展指令
- 移位指令
- 位操作指令
- 条件设置字节指令
- 循环控制指令
- 字符串操作指令

❖80486新增指令(12)
- 字节交换指令
- 交换并相加指令
- 比较并交换指令
- Cache管理指令

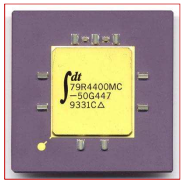
❖Pentium新增指令(289)
- 8字节比较交换指令
- 处理器特征识别指令
- 读时间标记计数器指令
- 读模型专用寄存器指令
- 写模型专用寄存器指令
- MMX(SIMD, 图形/音频)
- SSE(图形/视频/语音)
- SSE2

❖8086(89) -> Pentium(505)

第五讲：指令系统与MIPS汇编

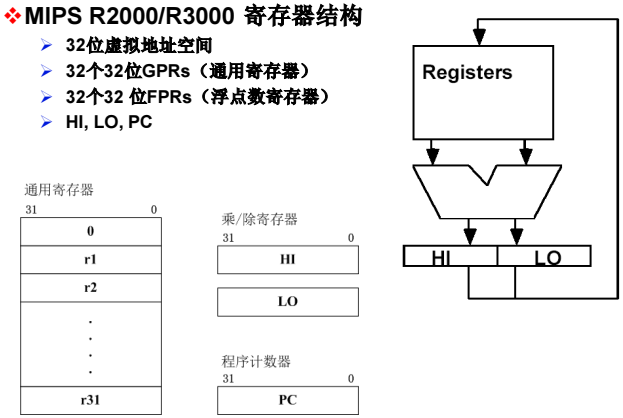
2.2 MIPS 指令格式简介

❖MIPS R系列CPU简介
- RISC（Reduced Instruction set Computer，精简指令集计算机）微处理器
- MIPS（Microprocessor without interlocked piped stages，无内部互锁流水级的微处理器），最早在80年代初由Stanford大学Patterson教授领导的研究小组研制出来，MIPS公司的R系列就是在此基础上开发的RISC微处理器。
- 1986年，推出R2000（32位）
- 1988年，推出R3000（32位）
- 1991年，推出R4000（64位）
- 1994年，推出R8000 （64位）
- 1996年，推出R10000
- 1997年，推出R20000
- 指令体系MIPS I、MIPS II、MIPS III、MIPS IV到MIPS V，嵌入式指令体系MIPS16、MIPS32到MIPS64的发展已经十分成熟。在设计理念上MIPS强调软硬件协同提高性能，同时简化硬件设计。



2.2 MIPS 指令格式简介

❖MIPS R2000/R3000 寄存器结构
- 32位虚拟地址空间
- 32个32位GPRs（通用寄存器）
- 32个32 位FPRs（浮点数寄存器）
- HI, LO, PC



2.2 MIPS 指令格式简介

❖MIPS 寄存器使用的约定

Name	Reg. Num	Usage
zero	0	constant value =0(恒为0)
at	1	reserved for assembler(为汇编程序保留)
v0 – v1	2 – 3	values for results(过程调用返回值)
a0 – a3	4 – 7	Arguments(过程调用参数)
t0 – t7	8 – 15	Temporaries(临时变量)
s0 – s7	16 – 23	Saved(保存)
t8 – t9	24 – 25	more temporaries(其他临时变量)
k0 – k1	26 – 27	reserved for kernel(为OS保留)
gp	28	global pointer(全局指针)
sp	29	stack pointer (栈指针)
fp	30	frame pointer (帧指针)
ra	31	return address (过程调用返回地址)

Registers are referenced either by number—$0…$31, or by name —$t0,$s1…$ra.

2.2 MIPS 指令格式简介

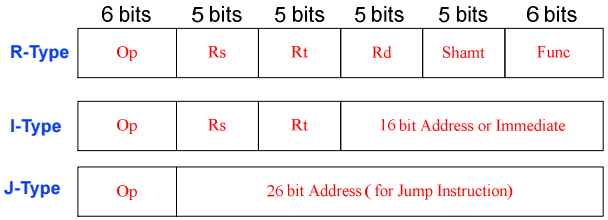
❖MIPS指令格式

- MIPS只有3种指令格式，32位固定长度指令格式
  - R-Type（Register类型）指令：两个寄存器操作数计算，结果送第三个寄存器
  - I-Type（Immediate类型）指令：使用1个16位立即数作；
  - J-Type（Jump类型）指令：跳转指令，26位跳转地址
- 最多3地址指令：add $t0, $s1, $s2 ($t0←$s1+$s2)
- 对于Load/Store指令，单一寻址模式：Base＋Displacement
- 没有间接寻址
- 16位立即数
- 简单转移条件（与0比较，或者比较两个寄存器是否相等）
- 无条件码

2.2 MIPS指令格式简介

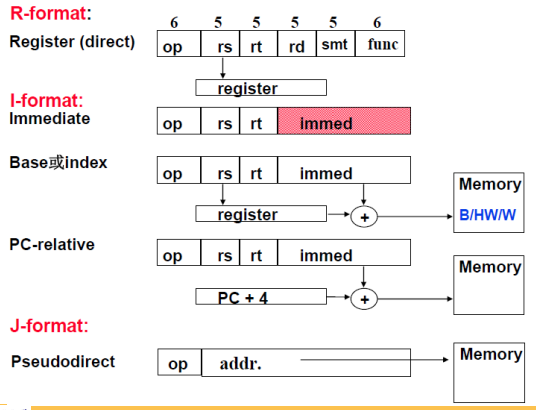
❖ MIPS 指令格式

- Op： 6 bits, Opcdoe
- Rs: 5 bits, The first register source operand
- Rt: 5 bits, The second register source operand
- Rd: 5 bits, The register destination operand
- Shamt: 5 bits, Shift amount (shift instruction)
- Func: 6 bits, function code (another Opcode)
  - R-Type指令OP字段为“000000”，具体操作由func字段给定



2.2 MIPS指令格式简介

❖MIPS 寻址方式



2.2 MIPS指令格式简介--指令类型

- Load/Store(取数/存储）指令
  - I-Type指令，存储器与通用寄存器之间传送数据
  - 支持唯一的寻址方式：Base+Index
  - 取数指令：LB（取字节）、LBU（取不带符号字节）、LH（取半字）、LHU（取不带符号的半字）、LW（取字）、LWL、LWR
  - 存储指令：SB（存字节）、SH（存半字）、SW（存字）、SWL、SWR
- 运算指令
  - R-Type指令（两个源操作数都是寄存器操作数）和I类型指令（一个源操作数是寄存器操作数，一个源操作数是16位立即数），目的操作数是寄存器。
  - ALU立即指令
  - 3操作数寄存器型指令
  - 移位指令
  - 乘/除指令

2.2 MIPS指令格式简介--指令类型

- 跳转和转移指令：控制程序执行顺序
  - 跳转指令：J-Type指令（26位绝对转向地址）或R类型指令（32位的寄存器地址）
  - 转移指令：I-Type指令，PC-relative寻址方式，相对程序计数器的16位位移量（立即数）。
  - 跳转：J、JAL、JR、JALR
  - 转移：BEQ（相等转移）、BNE（不等转移）、BLEZ（小于或等于0转移）、BGTZ（大于0 转移）、BLTZ（小于0转移）、BLTZAL、BGEZAL
- 特殊指令
  - R-Type指令
  - 系统调用SYSCALL
  - 断点BREAK

2.2 MIPS指令格式简介

- R-Type指令编码示例
  - 指令：add $t0, $s1, $s2 ; t0 ← (S1) + (s2)

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
指令格式	Op	Rs	Rt	Rd	Shamt	Func
指令编码	00000	10001	10010	01000	00000	10000

Rd ← (Rs) + (Rt)

- Op = 000000（表示R-Type)
- Func = 100000（表示add）
- Rs = 10001（表示s1）
- Rt = 10010（表示s2）
- Rd = 01000（表示t0）
- Shamt=00000（表示没有移位）

2.2 MIPS指令格式简介—指令示例

Instruction	*Example*	*Meaning*	*Comments*
add	add $1,$2,$3	$1← $2 + $3	3 operation
subtract	sub $1,$2,$3	$1← $2 – $3	3 operation
add immediate	addi $1,$2,100	$1← $2 + 100	+ constant
multiply	mult $2,$3	Hi,Lo← $2× $3	64-bit signed product
divide	div $2,$3	Lo ← $2 ÷ $3 Hi ← $2 mod $3	Lo = quotient Hi = remainder
move from Hi	mfhi $1	$1 ← Hi	Get a copy of Hi
move from Lo	mflo $1	$1 ← Lo	Get a copy of Lo
and	and $1,$2,$3	$1← $2 & $3	Logical AND
or	or $1,$2,$3	$1← $2 | $3	Logical OR
store	sw $3,500($4)	Mem($4+500)←$3	Store Word
load	lw $1,-30($2)	$1←Mem($2-30)	Load word
jump and link	jal 1000	$31=PC+4 Go to 1000	Procedure call
jump register	jr $31	Go to $31	procedure return
set on less than	slt $1,$2,$3	if ($2 < $3) than $1=1 else $1=0	

SWAP：MIPS过程示例

```
swap:
  addi   $sp,$sp, –12      ; Make room on stack for 3 registers
  sw     $31, 8($sp)       ; Save return address
  sw     $s2, 4($sp)       ; Save registers on stack
  sw     $s3, 0($sp)
  ....
```

```
sll     $s2, $a1, 2       ; mulitply k by 4
addu    $s2, $s2, $a0     ; address of v[k]
lw      $t0, 0($s2)       ; load v[k]
lw      $s3, 4($s2)       ; load v[k+1]
sw      $s3, 0($s2)       ; store v[k+1] into v[k]
sw      $t0, 4($s2)       ; store old v[k] into v[k+1]
```

```
  lw     $s3, 0($sp)       ; Restored registers from stack
  lw     $s2, 4($sp)
  lw     $31, 8($sp)       ; Restore return address
  addi   $sp,$sp, 12       ; restore top of stack
  jr     $31               ; return to place that called swap
```


第五讲：指令系统与MIPS汇编

2.3 CISC与RISC

❖指令系统优化设计的两种相反的方向

- 增强指令功能：CISC（Complex Instruction Set Computer），即复杂指令系统计算机
  - 特点：格式复杂，寻址方式复杂，指令种类多；把一些原来由软件实现的、常用的功能改用硬件的指令系统来实现。
  - 实例：80X86指令系统
- 简化指令功能：RISC（ Reduced Instruction Set Computer），即精简指令系统计算机
  - 特点：格式简单，指令长度和操作码长度固定；简单寻址方式，大部分指令使用寄存器直接寻址。
  - 实例：MIPS 指令系统

2.3 CISC与RISC

❖CISC的背景

- 计算机硬件成本的不断下降，软件开发成本的不断提高

  在指令系统中增加更多的、更复杂的指令，以提高操作系统的效率，并尽量缩短指令系统与高级语言的语义差别，以便于高级语言的编译。
- 程序的兼容性

  同一系列计算机的新机器和高档机的指令系统只能扩充而不能缩减。

❖CISC指令系统的特点

- 指令系统复杂庞大(一般数百条指令）；
- 寻址方式多，指令格式多，指令字长不固定；
- 可访存指令不受限制；
- 各种指令使用频率相差很大；
- 各种指令执行时间相差也很大；
- 大多数采用微程序控制器。

❖RISC的背景

- 80-20规律
  - 典型程序中80%的语句仅仅使用处理机中20%的指令，而且这些指令都是属于简单指令，如：取数、加、转移等。
  - 付出巨大代价添加的复杂指令仅有20%的使用概率
- VLSI时代
  - VLSI，即超大规模集成电路(Very Large Scale Integrated circuites)；
  - 复杂的指令系统需要复杂的控制器，占用较多的芯片面积，它的设计、验证、实现都变得更加困难。

❖RISC技术

- 把使用频率为80%的、在指令系统中仅占20%的简单指令保留下来，消除剩余80%的复杂指令，复杂功能用子程序实现。
- 不用微程序控制，采用简单的硬连线控制，控制器极大简化，加上优化编译配合硬件的改进，使系统的速度大大提高；
- 短周期时间、单周期执行指令（指令执行在一个机器周期内完成）；
- Load（取）/Store（存）结构，取数（存储器→寄存器）、存数（寄存器→存储器）
- 大寄存器堆，寄存器数量较多
- 哈佛（Harvard）总线结构，指令Cache、数据Cache，双总线动态访问机构；
- 高效的流水线结构、延迟转移、重叠寄存器窗口技术等

❖RISC的指令系统的特点

- 处理器通用寄存器数量较多；
- 由使用频率较高的简单指令构成；
- 简单固定格式的指令系统；
- 指令格式种类少，寻址方式种类少；
- 访问内存仅限Load/Store指令，其他操作针对寄存器；
- 指令采用流水技术。

2.3 CISC与RISC

- RISC与CISC性能对比
  - RISC比CISC机器的CPI（Cycles per Instruction，平均周期数）要小；
  - CISC一般用微码技术，一条指令往往要用好几个周期才能实现，复杂指令所需的周期数则更多，CISC机器CPI一般为4-6；
  - RISC一般指令一个周期完成，所以CPI=1，但LOAD、STORE等指令要长些，所以RISC机器的CPI约大于1。

- RISC与CISC技术的融合
  - 随着芯片集成度和硬件速度的增大，RISC系统也越来越复杂
  - CISC也吸收了很多RISC的设计思想
    - 例如：Inter 80486比80286更加注重常用指令的执行效率，减少常用指令执行所需的周期数。

第五讲：指令系统与MIPS汇编

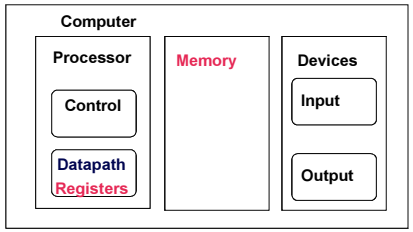
CPU和指令集

- 执行指令是CPU的主要工作
- 不同的CPU有不同的指令集
  - 指令集架构*Instruction Set Architecture* (*ISA*).
  - Intel 80x86 (Pentium 4), IBM/Motorola PowerPC (Macintosh), MIPS, Intel IA64, ...
- 精简指令集(RISC)的哲学
- MIPS – 最早一家生产出商用 RISC 架构的半导体公司
  - MIPS 简单、优雅，不被细节所累
  - MIPS 在嵌入式中广泛应用, x86 很少应用到嵌入式市场, 它更多的是应用到PC上

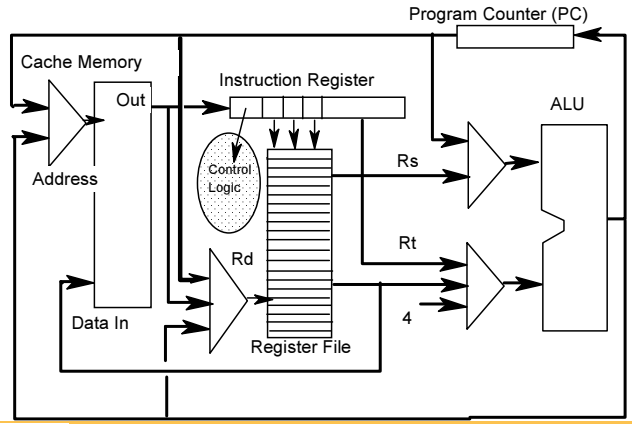
Most HP LaserJet workgroup printers are driven by MIPS-based™ 64-bit processors.

计算机系统的组成结构

- 计算机系统由4大部分组成
  - 控制
  - 运算
  - 存储
  - 输入输出
- 寄存器是数据通路的一部分



数据通路



寄存器堆

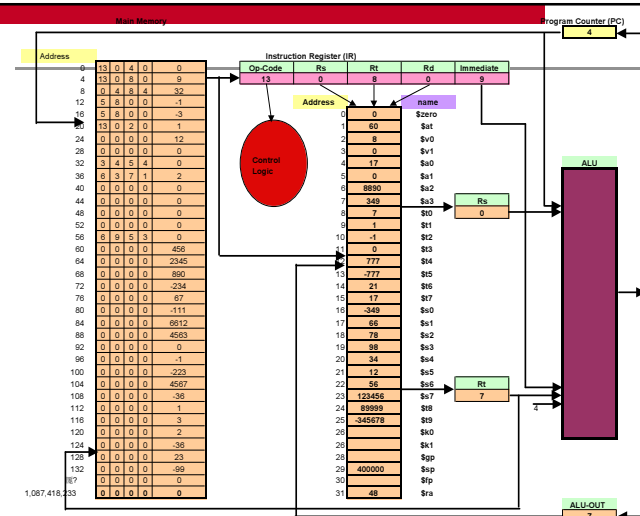
Number	Value	Name
0		$zero
1		$at
2		$v0
3		$v1
4		$a0
5		$a1
6		$a2
7		$a3
8		$t0
9		$t1
10		$t2
11		$t3
12		$t4
13		$t5
14		$t6
15		$t7
16		$s0
17		$s1
18		$s2
19		$s3
20		$s4
21		$s5
22		$s6
23		$s7
24		$t8

函数返回值

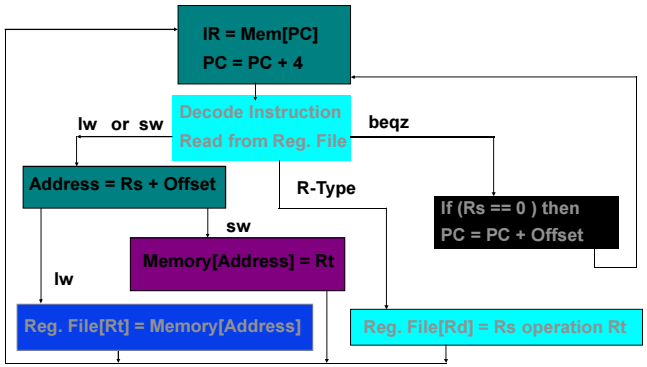
函数传递参数

临时变量寄存器

保存寄存器

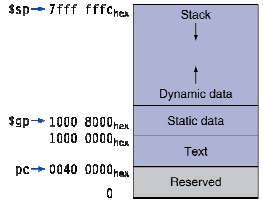


寄存器传送的控制逻辑

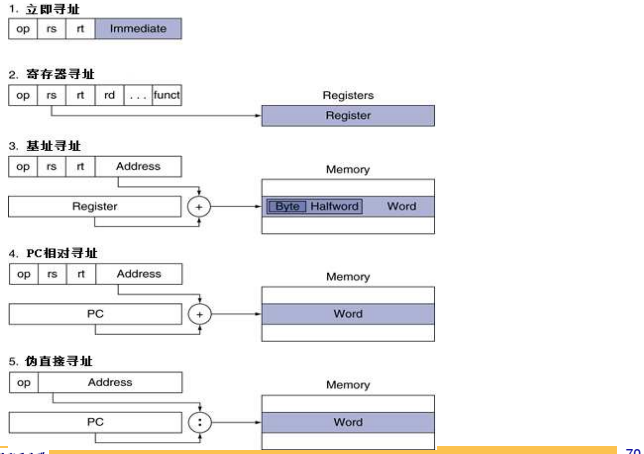


内存布局

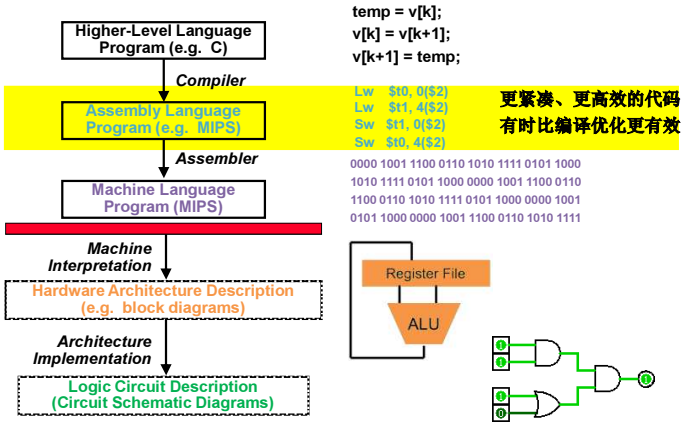
- Text: 程序代码段
- Static data: 全局变量
  - 例如, C语言中的静态变量, 常数数组和串
  - $gp 寄存器初始地址±偏移量寻址本段内存
- Dynamic data: 堆
  - 例如, C中的malloc, Java中的new
- Stack: 栈，自动存储区



寻址模式回顾



汇编语言在层次结构中的位置




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
Lw   $t0, 0($2)
Lw   $t1, 4($2)
Sw   $t1, 0($2)
Sw   $t0, 4($2)
```

更紧凑、更高效的代码
有时比编译优化更有效

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```


MIPS 汇编语言程序示例

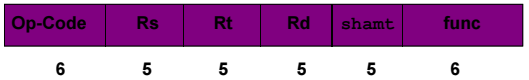
```
Label     Op-Code  Dest.   S1,     S2     Comments
          move     $a0,    $0             # $a0 = 0
          li       $t0,    99             # $t0 = 99
loop:
          add      $a0,    $a0,    $t0    # $a0 = $a0 + $t0
          addi     $t0,    $t0,    -1     # $t0 = $t0 - 1
          bnez     $t0,    loop           # if ($t0  != zero) branch to loop
          li       $v0,    1              # Print the value in $a0
          syscall
          li       $v0,    10             # Terminate Program Run
          syscall
```

MIPS指令集

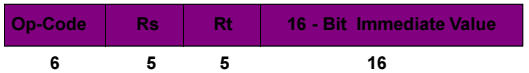
- 算术、逻辑和移位指令
- 存/取指令
- 条件分支指令
- 函数调用指令

MIPS指令字格式

- R- Format（ Register ）



- I- Format（ Immediate ）



- J- Format（Jump）

伪指令

- 取地址 `la $s0, table`
- 取立即数 `li $v0, 10`
- 移动 `move $t8, $sp`
- 乘 `mul $t2, $a0, $a1`
- 除 `div $s1, $v1, $t7`
- 求余 `rem $s2, $v1, $t7`
- 取反 `neg $s0, $s0`

MIPS 寄存器堆

寄存器命名规范约定

$0 ：常量0
$v0 ：函数返回值
$a0 ：函数传递参数
$t0 ：临时变量寄存器
$s0 ：保存寄存器
$sp ：栈指针
$ra ：返回地址

第五讲：指令系统与MIPS汇编

一．指令格式
1. 指令系统概述
2. 指令格式
3. 寻址方式

二．典型指令系统介绍
1. 8086/8088指令系统
2. MIPS指令系统
3. CISC与RISC

三．MIPS汇编语言
1. 概述
2. MIPS汇编指令和指令字
3. MIPS汇编程序

MIPS汇编指令和指令字

- MIPS指令语法、变量和注释
- MIPS汇编中的算术、逻辑和移位运算
- MIPS汇编中的数据存取
- MIPS汇编中的分支和循环
- MIPS汇编中的函数
- MIPS汇编中的寄存器规范约定
- 存储程序概念和MIPS指令字

MIPS 指令

- 指令语法
  操作符， 目标，源1，源2
  1) 操作名称（“操作符”）
  2) 操作结果（“目标”）
  3) 1st 操作数（“源1”）
  4) 2nd 操作数（“源2”）
- 语法是固定的
  - 1 个操作符, 3 个操作数
  - 通过约定好的统一的规则使硬件实现更简单
- 汇编语言中, 每一条语句就是执行指令集中的一条简单指令
- 与C语言 (以及其他大多数高级程序语言)不同, 每行汇编代码仅执行一条汇编指令
  - C 或 Java 等高级语言执行的指令数与所做的操作(=, +, -, *, /)有关

MIPS汇编变量: 寄存器（Registers） (1/2)

- 与高级程序设计语言C或Java不同, 汇编语言不使用变量
  - Why not? 保持硬件的简单
- 汇编语言的操作数都是（寄存器） registers
  - 直接在硬件上实现的，数量有限
  - 所有的操作都只能在寄存器上完成!
- 优势: 寄存器是直接用硬件实现的, 它们一定很快！
- 劣势: 寄存器是直接在硬件上实现的, 它们预先就规定好了固定的数目！
  - 解决: MIPS 代码必须仔细的写，以便能够合理、有效地使用寄存器
- MIPS有32个寄存器
  - 为什么32? 简单源自规整！
- 每一个MIPS寄存器的宽度为32bits
  - MIPS中，由32个二进制位组合在一起称为一个 字

MIPS汇编变量:寄存器（Registers） (2/2)

- 从0到31给32个寄存器编号
  - 每个寄存器除了有编号外，还有自己的名字
- 按照编号的引用方式:

```
$0, $1, $2, … $30, $31
```

- 约定：每个寄存器都取一个名字以便写代码时更方便

```
$16 - $23 ➔     $s0 - $s7
```

(与C语言中的变量对应)

```
$8 - $15, $24-$25 ➔     $t0 - $t7, $8-$9
```

(与临时变量对应)

- 通常情况下, 使用名字来指定寄存器，这样可以增加代码的可读性

C, Java 变量 vs. 寄存器

- C 语言(以及其他大多数高级程序语言) 中所有变量需要事先声明成为一个特定的类型
  - 例如:

```
int fahr, celsius;
char a, b, c, d, e;
```

- 所有变量*只能*表达它被声明的那个类型的一个值 (int与char用法不同，不能混淆)
- 在汇编语言里, *寄存器没有数值类型*; 只能通过代码来决定这些寄存器中的数值应该如何使用和处理

注释

- 另一个增加代码可读性的方法: 注释
- # 被用来做MIPS的注释
  - 任何从# 后开始到行末的内容都被视为注释内容，并被忽略
- 注意: 与C语言的区别
  - C 还有一种注释格式

```
/* comment */
```

这种方式可以跨越多行加注释

MIPS汇编指令和存储格式

- MIPS指令语法、变量和注释
- MIPS汇编中的算术、逻辑和移位运算
- MIPS汇编中的数据存取
- MIPS汇编中的分支和循环
- MIPS汇编中的函数
- MIPS汇编中的寄存器规范约定
- 存储程序概念和MIPS指令字

MIPS 整数加/减

- 加法
  - 例子: `add $s0,$s1,$s2` (in MIPS)，相当于: `a = b + c` (in C)
- 减法
  - 例子：`sub $s3,$s4,$s5` (in MIPS)，相当于：`d = e - f` (in C)
- C 语言中一条语句 `a = b + c + d - e`
  - 拆成多条汇编指令

```
add $t0, $s1, $s2 # temp = b + c
add $t0, $t0, $s3 # temp = temp + d
sub $s0, $t0, $s4 # a = temp - e
```

- C中每一行语句可以拆成许多行的MIPS
  - 编译时，每行"#"后面的语句都会被当成注释忽略
- 另一条C语言的语句：`f = (g + h) - (i + j)`
  - 使用临时变量寄存器

```
add $t0,$s1,$s2  # temp = g + h
add $t1,$s3,$s4  # temp = i + j
sub $s0,$t0,$t1  # f=(g+h)-(i+j)
```


寄存器 Zero和立即数

- 一个特别的立即数, 数字0, 在代码中频繁出现
- 定义寄存器 zero (`$0` or `$zero`)

```
add $s0,$s1,$zero (in MIPS)
  f = g (in C)
```

  - 考虑指令 `add $zero,$zero,$s0`
- 立即数是数值常量
  - 在代码中频繁出现，所以专门针对立即数定义了一些指令
- 立即数加

```
addi $s0,$s1,10 (in MIPS)
f = g + 10 (in C)
```

  - 语法与add指令类似，除了最后一个参数用数值代替了寄存器
- MIPS中没有立即数的减法
  - 尽可能地保持指令集越小越好

addi …, -X = subi …, X => 所以没有subi

例子：addi $s0,$s1,-10 (in MIPS)

f = g - 10 (in C)

算术运算中的溢出

- 发生溢出是由于计算机有限的数值表示引起的
- 例如（4位无符号数）

+15	1111	但是如果我们只有4位寄存器，存不了5位, 那么这个结果就是0010, 是+2, 结果就错了
+3	0011	
+18	10010	

- 有些语言会自动检测出异常(Ada), 有些不会(C)
- MIPS有2种(+,-)运算指令，每一种指令又有两种数值方式
  - 下面的 可以检测出溢出异常
    - add (`add`)
    - add immediate (`addi`)
    - subtract (`sub`)
  - 下面的 不会检测出溢出异常
    - add unsigned (`addu`)
    - add immediate unsigned (`addiu`)
    - subtract unsigned (`subu`)
- 编译器会自动挑选合适的运算指令类型
  - MIPS中的C编译器会使用 `addu, addiu, subu` 不检查溢出异常

位操作

- 把寄存器中的值拆开来，看成是32个单独的位，每个位表示一个2进制数值
  - 因为寄存器是由32个位组成的，有时候可能会想去单独处理其中某个或某些位，而非整个数值
  - 逻辑和移位操作

逻辑操作符

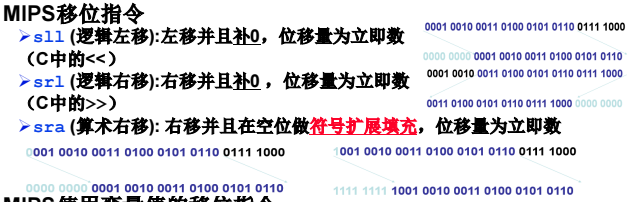
- 两种基本的逻辑操作符
  - AND: 当两个数都为1时输出1
  - OR: 至少有一个为1时输出1
- 逻辑指令的语法

  操作符　结果寄存器, 1st 操作数（寄存器）, 2nd 操作数（寄存器/立即数）
- 操作符（指令名）
  - and, or: 这两种指令的第三个参数（即2nd 操作数）都是寄存器
  - andi, ori: 这两种指令的第三个参数（即2nd 操作数）都是立即数
- 按位与、按位或
  - C: 按位与&，(比如, `z = x & y;`)
  - C: 按位或|，(比如, `z = x | y;`)

移位指令

- 将一个字的所有位向左或向右移动一定的位数
- 移位指令语法

  操作符　结果寄存器, 1st 操作数（寄存器）, 移位量（<32的常量/寄存器）
- MIPS移位指令
  - `sll` (逻辑左移):左移并且补0，位移量为立即数（C中的<<）
  - `srl` (逻辑右移):右移并且补0，位移量为立即数（C中的>>）
  - `sra` (算术右移): 右移并且在空位做符号扩展填充，位移量为立即数



- MIPS使用变量值的移位指令
  - `sllv, srlv, srav`
  - 与使用立即数位移量的处理方式类似, 移位量存储在寄存器中
- 好的编译器会在乘以2^n时，自动将乘法转换为移n位操作

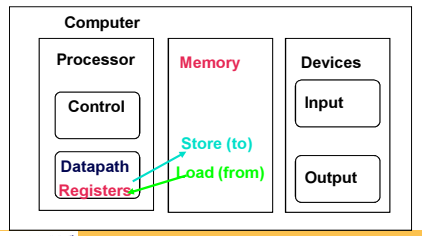
  `a *= 8;` (in C) → `sll $s0,$s0,3` (in MIPS)
- 类似的, 算术右移n位就是除以2^n

MIPS汇编指令和指令字

- MIPS指令语法、变量和注释
- MIPS汇编中的算术、逻辑和移位运算
- MIPS汇编中的数据存取
- MIPS汇编中的分支和循环
- MIPS汇编中的函数
- MIPS汇编中的寄存器规范约定
- 存储程序概念和MIPS指令字

MIPS汇编操作数: 内存

- C变量被映射成寄存器; 如果是大型数据结构，例如数组该怎么办呢?
  - MIPS 算术指令只能操作寄存器, 不能直接操作内存
- 数据存取指令在内存与寄存器之间传输数据
  - 内存到寄存器
  - 寄存器到内存



寄存器被包含在处理器的数据通路中; 如果操作数在内存中，我们必须首先将它们转移到处理器上才能进行操作，然后当我们处理完成之后，再将其送回内存

数据存取: 内存到寄存器

- 要指定访问内存的具体地址, 需要两个数据值
  - 一个指向内存某地址的指针和一个数字偏移量
  - 我们需要的内存地址通常是这两个值相加所得的和
- 例子: `8($t0)`
  - 根据`$t0`中的值找到内存中的对应地址, 再加上 8 bytes
- Load 指令语法

  操作码，寄存器，数值偏移量(寄存器)
- Load指令操作码：`lw`

例子: `lw $t0,12($s0)`

先取出$s0中的指针的值, 加上12 bytes 的偏移量, 将这两个值相加即得到我们要访问的内存地址，然后，将内存中该地址存储的数值送进$t0。

- `$s0` 称为基址寄存器，12 称为偏移量
- 偏移量经常用来访问数组或结构体中的元素: 基址寄存器指向该数组或结构体的首地址 (注意偏移量必须是一个 在编译时就已经有确定值的数值常量)

数据存取: 寄存器到内存

- 将寄存器中的数值写回内存中去
- Store与Load指令的语法格式是完全一样的

  操作码，寄存器，数值偏移量(寄存器)
- MIPS 指令名：`sw`

例子: `sw $t0,12($s0)`

先取出$s0中存放的指针的值, 加上12 bytes, 计算出两个值的和，即是要访问的内存地址，然后，将$t0中的值送入内存中该地址存放

- 记住: “Store 进内存”

指针 vs. 值

- 一个寄存器中可以存储32-bit的数值
  - 它可以是一个 (signed) `int`, 一个`unsigned int`, 一个 pointer (内存地址), 或是其它的类型
  - 如果你写了`add $t2,$t1,$t0` 那么 `$t0` 与 `$t1`最好是可以相加的，结果是有意义的值
  - 如果你写了`lw $t2,0($t0)` 那么 `$t0` 里面一定要是一个地址值(pointer)

寻址: 字节 vs. 字

- 内存中的每一个字都有一个地址, 类似于一个数组的索引
- 早期的计算机按字编址，就像C语言中的数组下标
  - `Memory[0], Memory[1], Memory[2], …`

被称作“字”地址

- 访问字(4 bytes/word)的同时也需要访问8-bit 字节(byte)
- 今天计算机按字节编址，32-bit (4 bytes) 字地址按 4递增
  - `Memory[0], Memory[4], Memory[8], …`
- 如果要选择C语言中的`A[5]`

`g = h + A[5]; g: $s1, h: $s2, $s3: A的基地址`

- 首先，将数据从内存送到寄存器:

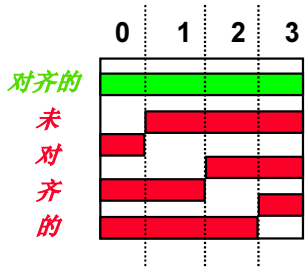
```
lw    $t0,20($s3)   # $t0 gets A[5]
```

- 然后，将它与h相加，结果送进g

```
add $s1,$s2,$t0   # $s1 = h+A[5]
```

字对齐

- 字节地址和字地址
  - 无论是 lw 还是 sw, 基址寄存器的值与偏移量的和始终应该是4的倍数
- MIPS 要求所有字的首地址必须是4的倍数
  - Alignment(字对齐): 对象的起始地址一定要是字长的整数倍



地址的最后一个16进制数:

0, 4, 8, or C_{hex}

1, 5, 9, or D_{hex}

2, 6, A, or E_{hex}

3, 7, B, or F_{hex}

寄存器vs. 内存

- 变量数比寄存器数多怎么办?
  - 编译器会将最经常使用的变量保留在寄存器中
  - 不常使用的放在内存中: spilling
- 为什么不把所有的变量都放在寄存器里?
  - 小就是快:
    寄存器比内存要快
  - MIPS 计算型指令每执行一条指令时，可以从两次读操作取数据，计算结果，一次写操作到寄存器中
  - MIPS 数据迁移型指令在每一条指令中，对寄存器只有一次读或写操作

例子1

如果把C语言中的表达式*x = *y 翻译成 MIPS汇编的指令，怎么样才是正确的呢?
(假定x, y 指针分别存储在 $s0, $s1中)

```
A: add $s0,   $s1, zero
B: add $s1,   $s0, zero
C: lw  $s0, 0($s1)
D: lw  $s1, 0($s0)
E: lw  $t0, 0($s1)
F: sw  $t0, 0($s0)
G: lw  $s0, 0($t0)
H: sw  $s1, 0($t0)
```

```
0: A
1: B
2: C
3: D
4: E→F
5: E→G
6: F→E
7: F→H
8: H→G
9: G→H
```


例子2

- 我们需要完成以下指令:

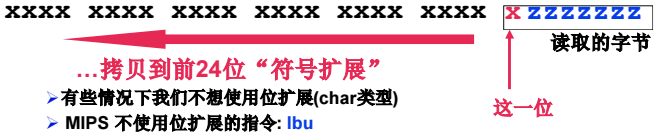
```
int x = 5;
*p = *p + x + 10;
```

- MIPS (假定 $s0 中存放 p, $s1等于x)

```
addi $s1,$0,5          # x = 5
lw   $t0,0($s0)        # temp = *p
add  $t0,$t0,$s1       # temp += x
addi $t0,$t0,10        # temp += 10
sw   $t0,0($s0)        # *p = temp
```

字节的存/取

- 除了需要在内存与寄存器之间按字传送 (lw, sw)外, MIPS 还有按字节传送的指令:
  - 读字节: lb
  - 写字节: sb
- 与lw, sw格式相同
  - 例如：lb $s0, 3($s1)
  - *把内存中的某个地址("3" + $s1中的地址值) 所存储的数值拷贝到$s0的低地址字节上.*
- 那么32位中的其余24位填充什么呢?
  - lb: 使用位扩展(或称为符号扩展)填充剩余24位



  - 有些情况下我们不想使用位扩展(char类型)
  - MIPS 不使用位扩展的指令: lbu

MIPS汇编指令和指令字

- MIPS指令语法、变量和注释
- MIPS汇编中的算术、逻辑和移位运算
- MIPS汇编中的数据存取
- MIPS汇编中的分支和循环
- MIPS汇编中的函数
- MIPS汇编中的寄存器规范约定
- 存储程序概念和MIPS指令字

- 目前为止，我们见过的指令都是操作数据的指令…我们做成了一个计算器
- 为了实现一个真正的计算机, 我们需要代码能够做一些判断、决定和转跳…
- C 和 MIPS都提供 labels标签 用来支持“goto” 在代码中跳来跳去.
  - C: 使用goto是非常可怕的; MIPS: 必须有goto!

C 判断: if 语句

- C中有两种if语句
  - if (*condition*) *clause*
  - if (*condition*) *clause1* else *clause2*
- 我们对上面第二种if语句改写如以下

```
if  (condition) goto L1;
     clause2;
     goto L2;
L1: clause1;
L2:
```

- 不如if-else看上去优雅, 但是效果一样

MIPS 判断指令

- MIPS中的判断指令
  - `beq  register1, register2, L1`
  - `beq` = “Branch if (registers are) equal”
    相当于C当中:
    `if (register1==register2) goto L1`
- MIPS中还有一个与其互补的指令
  - `bne  register1, register2, L1`
  - `bne` = “Branch if (registers are) not equal”
    相当于C当中:
    `if (register1!=register2) goto L1`
- 称为条件分支

MIPS ‘Goto’ 指令

- 除了条件分支语句之外, MIPS 还有无条件分支

```
j    label
```

  - 跳转指令: 跳转到标签label所在的代码，不需要满足任何条件
- 与C中的goto语句用法相同

```
goto label
```

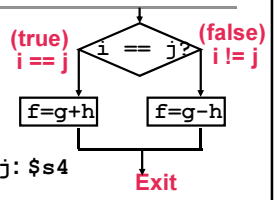
- 从技术的角度来说, 它与以下写法相同

```
beq    $0,$0,label
```


从C的if语句到MIPS

- C语言中的条件判断

```
if (i == j) f=g+h;
else f=g-h;
```



- 假定使用如下的映射

f: $s0 ; g: $s1; h: $s2; i: $s3; j: $s4

- MIPS代码

```
    beq $s3,$s4,True   # branch i==j
    sub $s0,$s1,$s2    # f=g-h(false)
    j   Fin            # goto Fin
True: add $s0,$s1,$s2    # f=g+h (true)
Fin:
```

注意: 编译器会在处理判断语句时自动为语句添加label去执行分支，所以我们在C的if语句中找不到goto和label

循环

- C中的循环; 假定A[] 是整型数组

```
do {
     g = g + A[i];
     i = i + j;
  } while (i != h);
```

重写成

```
Loop:  g = g + A[i];
       i = i + j;
       if (i != h) goto Loop;
```

- 假定使用以下映射

g, h, i, j, A[]的基址分别对应$s1,$s2,$s3,$s4,$s5

- MIPS 代码

```
Loop: sll $t1,$s3,2    #$t1= 4*i
      add $t1,$t1,$s5  #$t1=addr A
      lw  $t1,0($t1)   #$t1=A[i]
      add $s1,$s1,$t1  #g=g+A[i]
      add $s3,$s3,$s4  #i=i+j
      bne $s3,$s2,Loop# goto Loop if i!=h
```

- C中有3种循环结构
  - While；do…while；for
  - 每一种都可以用另外两种中的任一种等价表达, 上面的例子同样适用于while 和 for 循环

MIPS中的不等式

- MIPS 不等式指令
  - 语法: `slt reg1,reg2,reg3`
  - 含义: “Set on Less Than” （‘set’ 指置为1）

```
if (reg2 < reg3)
      reg1 = 1;        reg1 = (reg2 < reg3);
else reg1 = 0;
```

- 如何表达这样的语句：`if (g < h) goto Less;`
- MIPS 代码（假定 g:$s0, h:$s1）

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # goto Less
                # if $t0!=0
                # (if (g<h)) Less:
```

- 寄存器 $0 的值总是 0, 所以 bne 和 beq 通常在slt指令后用寄存器0来做分支判断
  使用slt ➔ bne 指令对表示if(... < ...)goto...
- 如何表达>, ≥? 增加新指令？

不等式中的立即数和无符号数

- slt 指令的立即数版本: slti

C
```
if (g >= 1) goto Loop
```

MIPS
```
Loop:     . . .
slti $t0,$s0,1     # $t0 = 1 if
                   # $s0<1 (g<1)
beq  $t0,$0,Loop   # goto Loop
                   # if $t0==0
                     # (if (g>=1))
```

使用slt ➔ beq 指令对表示 if(... ≥ ...)goto...

- 无符号数不等式指令: sltu, sltiu

基于无符号数比较的结果置位

MIPS 有符号 vs. 无符号

- 比较以下指令执行后$t0和$t1的值
  ($s0=FFFFFFFAH , $s1=0000FFFAH)

```
slt $t0, $s0, $s1
sltu $t1, $s0, $s1
```

- MIPS 中的有符号和无符号
  - 有符号扩展/无符号扩展
    (lb, lbu)
  - 没有(不得)溢出
    (addu, addiu, subu, multu, divu)
  - 有符号比较/无符号比较
    (slt, slti/sltu, sltiu)

例子: C语言 Switch 语句

- 根据k的取值从4个选项中选择1个，C代码如下：

```
switch (k) {
 case 0: f=i+j; break; /* k=0 */
 case 1: f=g+h; break; /* k=1 */
 case 2: f=g-h; break; /* k=2 */
 case 3: f=i-j; break; /* k=3 */
}
```

- 先简化成if-else语句链

```
if(k==0) f=i+j;
 else if(k==1) f=g+h;
   else if(k==2) f=g-h;
     else if(k==3) f=i-j;
```

变量映射: f:$s0, g:$s1, h:$s2, i:$s3, j:$s4, k:$s5

- MIPS 代码如下

```
        bne $s5,$0,L1    # branch k!=0
        add  $s0,$s3,$s4 #k==0 so f=i+j
        j    Exit        # end of case so Exit
L1:     addi $t0,$s5,-1  # $t0=k-1
        bne  $t0,$0,L2   # branch k!=1
        add  $s0,$s1,$s2 #k==1 so f=g+h
        j    Exit        # end of case so Exit
L2:     addi $t0,$s5,-2  # $t0=k-2
        bne  $t0,$0,L3   # branch k!=2
        sub  $s0,$s1,$s2 #k==2 so f=g-h
        j    Exit        # end of case so Exit
L3:     addi $t0,$s5,-3  # $t0=k-3
        bne  $t0,$0,Exit # branch k!=3
        sub  $s0,$s3,$s4 #k==3 so f=i-j
Exit:
```

MIPS汇编指令和指令字

- MIPS指令语法、变量和注释
- MIPS汇编中的算术、逻辑和移位运算
- MIPS汇编中的数据存取
- MIPS汇编中的分支和循环
- MIPS汇编中的函数
- MIPS汇编中的寄存器规范约定
- 存储程序概念和MIPS指令字

C 函数

```
main() {
  int i,j,k,m;
  ...
  i = mult(j,k); ...
  m = mult(i,i); ...
}
/* really dumb mult function */
int mult (int mcand, int mlier){
  int product;
 product = 0;
  while (mlier > 0)  {
   product = product + mcand;
   mlier = mlier -1; }
  return product;
  }
```

哪些信息是编译器和程序员需要追踪和记录的？

什么指令可以实现这样的功能？

记录函数调用

- 寄存器用来记录函数调用信息
- *寄存器规范*
  - 返回地址 $ra
  - 参数 $a0, $a1, $a2, $a3
  - 返回值 $v0, $v1
  - 局部变量 $s0, $s1, … , $s7
- 还会用到栈！

支持函数功能的指令(1/3)

```
C  ... sum(a,b);... /* a,b:$s0,$s1 */
   }
   int sum(int x, int y) {
       return x+y;
   }
```

MIPS

address
1000
1004
1008
1012
1016
2000
2004

MIPS中,所有指令都占4个字节,并且像数据一样存在内存中（存储程序概念），这里的地址是程序存储的地址

支持函数功能的指令(2/3)

C
```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

MIPS
```
address
1000  add  $a0,$s0,$zero   # x = a
1004  add  $a1,$s1,$zero   # y = b
1008  addi $ra,$zero,1016 #$ra=1016
1012  j    sum             #jump to sum
1016  ...
2000  sum: add $v0,$a0,$a1
2004  jr   $ra             # new instruction
```

为什么用jr而不用 j?

sum 可能会被多个函数调用，不能返回到某个固定的地址！
调用的函数一定会指定返回的地址

支持函数功能的指令(3/3)

- 可同时执行转跳和存储返回地址的指令
  - jal，(jump and link)

```
1008 addi $ra,$zero,1016 #$ra=1016
1012 j sum                #goto sum
```

可用单条指令代替

```
1008 jal sum  # $ra=1012,goto sum
```

- 用jal使得函数调用更快，同时不需要了解代码读入内存的地址细节
- jal 与 j 的语法相同

```
jal   label
```

- jal 执行步骤其实应该是‘laj’ (link and jump)
  - 第一步 (link): 将 下一条指令地址存入$ra (为什么下一条?)
  - 第二步 (jump): 向给定的label转跳
- 寄存器转跳指令jr，可转跳至寄存器中存储的地址

```
jr register
```

  - 在函数调用中非常有用

jal 指令将返回地址存储在寄存器($ra)中
jr $ra 跳回该地址

嵌套调用

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y;
}
```

- sumSquare被调用，而sumSquare又调用mult
  - $ra中存储着 sumSquare 的返回地址，但是会在调用 mult时重写
  - 需要在调用mult 之前存储 sumSquare 返回地址
- 需要在$ra之外存储相关信息！
- 当一个 C 程序运行时，有3块重要的内存区域被分配
  - 静态区（static）：存储静态变量，一旦程序声明，直到程序执行结束才会清除，比如C程序的全局变量
  - 堆（heap）：动态声明的变量
  - 栈（stack）：程序执行过程中使用的空间，可用来存储寄存器值

使用栈

- 寄存器$sp始终指向栈空间最后被使用的位置—栈指针
- 使用栈的时候，对该指针减去需要的空间量，并向该空间填写信息
- 刚才的C例子

```
int sumSquare(int x, int y){
    return mult(x,x)+ y;}
```

```
sumSquare:
"push"  addi $sp,$sp,-8 # space on stack
        sw $ra, 4($sp)      # save ret addr
        sw $a1, 0($sp)      # save y

        add $a1,$a0,$zero # prep args
        jal mult              # call mult

        lw $a1, 0($sp)      # restore y
"pop"   add $v0,$v0,$a1 # mult()+y
        lw $ra, 4($sp)      # get ret addr
        addi $sp,$sp,8  # restore stack
        jr $ra
mult: ...
```

过程调用规则

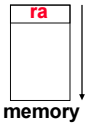
- 过程调用的步骤
  - 将需要保存的值压入栈
  - 如果需要的话，指定参数
  - jal 调用
  - 从栈中恢复相关的值
- 调用过程中的规则
  - 通过 jal 指令调用，使用一个 jr $ra 指令返回
  - 最多可接受4个入口参数， $a0, $a1, $a2, $a3
  - 返回值通常在 $v0 中(如果需要，可以使用$v1)
  - 必须遵守寄存器使用规范(即使是在那些只有你自己调用的函数中)!

一个函数的基本结构

```
entry_label:
        addi $sp,$sp, -framesize
        sw $ra, framesize-4($sp)   # save $ra
        save other regs if need be

        Body . . . (call other functions…)

        restore other regs if need be
        lw $ra, framesize-4($sp)   # restore $ra
        addi $sp,$sp, framesize
        jr $ra
```



MIPS汇编指令和指令字

- MIPS指令语法、变量和注释
- MIPS汇编中的算术、逻辑和移位运算
- MIPS汇编中的数据存取
- MIPS汇编中的分支和循环
- MIPS汇编中的函数
- MIPS汇编中的寄存器规范约定
- 存储程序概念和MIPS指令字

MIPS 寄存器分配

寄存器 0	$0	$zero
汇编器预留	$1	$at
返回值	$2-$3	$v0-$v1
参数	$4-$7	$a0-$a3
临时	$8-$15	$t0-$t7
保存	$16-$23	$s0-$s7
临时	$24-$25	$t8-$t9
内核占用	$26-27	$k0-$k1
全局指针	$28	$gp
栈指针	$29	$sp
帧指针	$30	$fp
返回地址	$31	$ra

- $at: 编译器随时可能使用，最好不要用
- $k0-$k1: 操作系统随时会使用，最好不要用
- $gp, $fp: 可以不用理会

寄存器规范 (1/2)

- CalleR: 发起调用的函数
- CalleE: 被调用的函数
- 当callee需要返回时, caller 需要知道哪些寄存器的值可能已经被改了，还有哪些是保证不更改的
- 寄存器规范: 一套被普遍遵从的规则——在执行了一个函数调用(jal) 后，哪些寄存器的值要保证不变，以及哪些可能已经变了
- 保存寄存器
  - $0: 不能改变， 永远是0
  - $s0-$s7: 如果被修改了需要恢复。如果 callee 由于各种原因改变了这些值, 它必须在返回之前将这些寄存器的原始值恢复
  - $sp:如果被修改了需要恢复. 栈指针在jal 执行之前和之后必须是指向的同一个地址, 如若不然 caller就无法从栈上正常恢复数据了
  - HINT – 所有保存寄存器都以 S开头！

寄存器规范 (2/2)

- 易变寄存器
  - $ra: 会改变。jal 会自动更改这个寄存器的值，Caller 需要将其值保存在栈上
  - $v0-$v1:会改变。始终保存最新的返回值
  - $a0-$a3:会改变。这几个都是易变的参数寄存器，Caller 如果在调用完Callee后还需要用到这些寄存器中的值，就需要在调用Callee前将这些值保存在自己的栈空间内
  - $t0-$t9:会改变。 任何函数在任何时候都可以更新这些寄存器中的值，Caller 如果在调用完Callee后还需要用到这些寄存器中的值，就需要在调用Callee前将这些值保存在自己的栈空间内
- 这样的规范和约定意味着什么？
  - 如果函数 R 调用函数 E, 那么函数 R 必须赶在调用（jal）执行前，将所有Callee可能会更新、而自己还会再用到的寄存器的值保存在自己的栈空间内
  - 如果非得用到的话，函数 E 必须保存所有它将要更新的 S (保存) 寄存器的值，并在函数返回（jr $ra）前及时恢复
- 调用和被调用都只需要保存他们使用的临时和保存寄存器值，并非所有的寄存器都要保存

MIPS汇编指令和指令字

- MIPS指令语法、变量和注释
- MIPS汇编中的算术、逻辑和移位运算
- MIPS汇编中的数据存取
- MIPS汇编中的分支和循环
- MIPS汇编中的函数
- MIPS汇编中的寄存器规范约定
- 存储程序概念和MIPS指令字

存储程序概念

- 冯诺依曼计算机建立在 2 个大原则之上
  - 指令与数值的表示形式一模一样
  - 全部程序可以被存储在内存中，像数据一样被读写
- 简化计算机系统的软/硬件
  - 用于数据操作的内存技术完全适用于指令操作
- 导致的结果1：编址
  - 所有的存储在内存中的东西都有一个地址，分支与跳转语句的执行正是基于此（C中的指针存的正是内存中的地址值）
  - 对地址的随意使用会导致很难查找的bug
  - 有一个寄存器始终保存正在执行的指令地址: “Program Counter” (PC)，从根本上说就是一个指向内存的指针
- 导致的结果2：二进制代码兼容性
  - 程序以二进制的形式给出，程序与特定的指令集绑定
  - 新机器想要运行旧程序 (“二进制代码”) 时，必须将程序按照新的指令集进行编译
  - 导致 “向后兼容” 的指令集不断进化

作为指令的数字

- 现在我们处理的所有的数据都是按字来分配的(32位字长)
  - 每个寄存器是一个字
  - lw, sw 每次只能访问内存中的一个字
- 如何来表示指令呢?
  - 计算机只认识1和0, 所以“add $t0,$0,$0”对计算机来说没有意义
  - MIPS 追求简单: 数据是按字存放的, 指令也按字存放吧!
- 一个字有32位, 我们把一个字分成几个“字段”(“fields”)
  - 每个“字段”用来提供指令的一部分信息
- 可以定义不同的分配“字段”的方法, MIPS 基于简单原则, 定义了以下3种指令格式的基本类型
  - I-format(立即数格式)
    当指令中有立即数的时候使用, 包括lw、sw(偏移量是立即数) 以及分支语句(beq and bne)。(但是这种格式不包含“移位”指令)
  - J-format(跳转指令格式)
    j , jal
  - R-format(寄存器格式)
    适用于其他的指令

R-Format 指令

- 以位为单位定义各个“字段”的 大小

Opcode(6)	rs(5)	rt(5)	rd(5)	shamt(5)	funct(6)

每个字段都被看成是5-bit（0-31）或6-bit（0-63）的无符号整数，而不是一个32-bit整数的一部分

- opcode: 与其他字段结合决定指令(等于0时代表所有R-Format指令)
- funct: 与opcode组合起来, 决定该条指令名(操作符)
- rs (Source Register): *通常*指定存放第一个操作数的寄存器
- rt (Target Register): *通常*指定存放第二个操作数的寄存器
- rd (Destination Register): *通常*指定存放计算结果的寄存器
- shamt: 这个字段中存储执行移位运算时要移的位数(该字段在不进行移位操作的指令中通常会置0)

注意3个寄存器字段:

- 每个寄存器字段是5-bit, 可以用它来完整的表示出0-31之间的所有无符号整数，这样每一个寄存器字段中的数值就是对应的32个寄存器中的一个
- 当然有些特殊情况，我们会在后面提到

R-Format 指令的例子

- MIPS 指令

```
add   $8,$9,$10
```

opcode = 0
funct = 32
rd = 8 (目标结果)
rs = 9 (第一 *操作数*)
rt = 10 (第二 *操作数*)
shamt = 0 (非移位指令)

每个字段的十进制表示

0	9	10	8	0	32

每个字段的二进制表示

000000	01001	01010	01000	00000	100000

hex

十六进制表示: $012A\ 4020_{16}$

十进制表示: $19,546,144_{10}$

称为 机器语言指令

I-Format 指令

- 带立即数的指令?
  - 5-bit 的字段只能表示最大31的整数值: 立即数有可能大得多
  - 如果指令中需要立即数的话, 执行这条至少需要2个寄存器

I-Format 指令

opcode(6)	rs(5)	rt(5)	immediate(16)

只有一个字段与R-format不同, opcode 还在原来的位置不变

- opcode: 因为没有了funct字段, opcode 在I-format指令中可以唯一确定一条指令 (R-format 用2个6-bit的字段而非一个12-bit字段来确定一条指令的原因: 为了与其他指令格式保持一致)
- rs: 表示*唯一*的操作数寄存器(如果有的话)
- rt: 存储计算结果的寄存器(*target* register)
- 立即数字段
  - ✓ addi, slti, sltiu, 立即数通过位扩展(符号扩展)的方式扩成32 位
  - ✓ 16 bits ➔ 可以表示出2^{16}个不同的整数值
  - ✓ 这么大的立即数在处理一些特别的指令(如lw 或 sw) 时已经足够了, 即使用slti指令，在大多数情况下也是没有问题的

I-Format 指令的例子

❖MIPS 指令

```
addi    $21,$22,-50
```

opcode = 8
rs = 22 (保存操作数的寄存器)
rt = 21 (目标寄存器，存储结果值用)
immediate = -50 (默认为十进制)

十进制表示

8	22	21	-50

二进制表示

001000	10110	10101	1111111111001110

十六进制表示: $0x22D5\ FFCE_{16}$
十进制表示: $584,449,998_{10}$

I-Format 指令的问题

❖立即数太大怎么办？
- 当需要的立即数在其字段内可以表示的时候，addi, lw, sw 和 slti 指令执行时都没有问题
- 但是如果太大，字段无法表示怎么办?在使用任何一个I-Format指令时，我们都必须考虑：如果立即数是一个32-bit的数值该怎么办？

❖解决方案
- 使用软件技巧 + 新的指令
- 不改变现有指令: 只要加入一条新指令来帮忙

❖新指令：lui register, immediate
- Load Upper Immediate
- 将一个16-bit的立即数存入寄存器的高16位，将寄存器的低16位全部置0

这样就没问题啦

例子: addi $t0,$t0, 0xABABCDCD
改为: lui $at, 0xABAB
ori $at, $at, 0xCDCD
add $t0,$t0,$at

立即数太大，这条指令根本放不进去

- 每条I-format指令只有16-bit用来存放立即数

哪一条指令的机器码与十进制数35长得一样?

1. add $0, $0, $0	0	0	0	0	0	32
2. subu $s0,$s0,$s0	0	16	16	16	0	35
3. lw $0, 0($0)	35	0	0	0		
4. addi $0, $0, 35	8	0	0	35		
5. subu $0, $0, $0	0	0	0	0	0	35

注意：只是长得一样，但是指令不是数字！！

使用I-Format的分支语句: 程序计数器相对寻址

opcode	rs	rt	immediate

- opcode 指明指令是beq或bne
- rs 和 rt 指明要比较的两个寄存器
- 立即数字段?
  - 立即数只有16 bits，PC (程序计数器) 有32-bit 的指向内存的指针；立即数无法表示出完整的内存地址
- 通常使用的if-else, while, for等分支语句，一般循环体都较小
- 函数调用与无条件跳转指令都会用到跳转指令(j and jal), 不是分支指令
- 结论:多数情况下，分支语句转跳时，PC 的变化值都相差不大
- 在32-bit 指令格式中执行分支语句的解决方案: PC-相对寻址
  - 将16-bit立即数使用补码表示，在需要分支的时候与PC *相加*
  - 可以分支到PC ± 2^{15} 字节的地方
  - 还能不能更好?
- 注意: 指令的起始地址都一定要是4的倍数(字对齐)
  - 和PC相加的立即数也应该是4的倍数！其实可以分支到PC ± 2^{15} *指令字* 了 (或者说 PC ± 2^{17} 字节)
  - PC = (PC + 4) + (immediate * 4)

分支的例子

❖MIPS 代码

```
Loop: beq   $9,$0,End
      add   $8,$8,$10
      addi  $9,$9,-1
      j     Loop
End:
```

1
2
3

beq 分支指令是I-Format格式的
- opcode = 4
- rs = 9 (第一操作数)
- rt = 0 (第二操作数)
- immediate = ???

立即数字段:

要和PC相加(或相减)的指令数, 是从分支语句的下一条指令算起的

在这条 beq 的分支中, immediate = 3

分支指令的十进制表示

4	9	0	3

分支指令的二进制表示

000100	01001	00000	0000000000000011

关于PC-相对寻址的问题

❖如果代码移动了，分支字段的值会变么?

❖如果目标地址与分支指令相差 > 2^{15}怎么办?

❖我们为什么需要这么多寻址方式? 一个不行么?

J-Format 指令

❖ 在分支语句中, 假定不会分支到太远的地方, 所以可以指明PC的*变化值*

❖ 对于一般的跳转指令 (j 和 jal), 是有可能跳到内存中*任意*一个地方的

➢理想情况下, 可以直接给出一个32-bit的内存地址，告诉要跳到哪里

J-Format 指令

opcode(6)	target address(26)

➢ 保持 opcode 字段与 R-format 及 I-format 一样，维护一致性原则

➢ 把其他所有字段都加到一起，使能表示的地址尽量大

➢ 利用字对齐，可以表示出32-bit地址的28 bits

➢ 剩下的最高4位根据定义, 直接从PC取

New PC = { PC[31..28], target address, 00 }

➢ 如果确实需要一个32-bit 地址, 就把它放进寄存器，使用jr指令

第五讲：指令系统与MIPS汇编

- MIPS汇编语言语句
- MIPS汇编语言程序模板
- 数据定义
- 内存对齐和字节序
- 系统调用
- 过程
- 参数传递和运行时栈

汇编语言语句

- MIPS汇编中的3类语句
  - 通常一个语句一行

1. 可执行指令
   - 为处理器生成在运行时执行的机器码
   - 指令告诉处理器该做什么
2. 伪指令和宏
   - 由汇编程序翻译成真正的指令
   - 简化编程人员的工作
3. 汇编伪指令
   - 当翻译代码时为汇编程序提供信息
   - 用来定义段、分配内存变量等
   - 不可执行: 汇编伪指令不是指令集的一部分

指令

- 汇编语言指令格式

  [标签:] 操作符 [操作数] [#注释]
- 标签: (可选)
  - 标记内存地址, 必须跟冒号
  - 通常在数据和代码段出现
- 操作符
  - 定义操作 (比如 add, sub, 等)
- 操作数
  - 指明操作需要的数据
  - 操作数可以是寄存器，内存变量或常数
  - 大多数指令有3个操作数

```
L1:    addiu $t0, $t0, 1          #increment $t0
```


注释

- 注释是非常重要的!
  - 解释程序语句的目的
  - 程序语句的编写、修改时间和人
  - 解释程序中得数据，输入和输出
  - 解释指令序列和算法
  - 每个过程的开始需要注释
    - 指出输入参数和过程的结果
    - 描述过程的功能
- 单行注释
  - 由‘#’开头在1行内结束

- MIPS汇编语言语句
- MIPS汇编语言程序模板
- 数据定义
- 内存对齐和字节序
- 系统调用
- 过程
- 参数传递和运行时栈

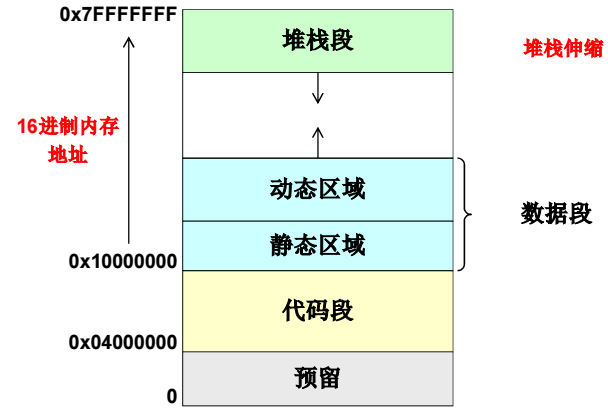
程序模板

```
# Title:                          Filename:
# Author:                         Date:
# Description:
# Input:
# Output:
################## Data segment ######################
.data
  . . .
################## Code segment ######################
.text
.globl main
main:                             # main program entry
  . . .
li $v0, 10                        # Exit program
syscall
```


.DATA, .TEXT, 和 .GLOBL 伪指令

- .DATA 伪指令
  - 定义程序的数据段
  - 程序的变量需要在该伪指令下定义
  - 汇编程序会分配和初始化变量的存储空间
- .TEXT 伪指令
  - 定义程序的代码段
- .GLOBL 伪指令
  - 声明一个符号为全局的
  - 全局符号可以被其它的文件引用
  - 用该伪指令声明一个程序的 *main* 过程

程序在内存中的存放



- MIPS汇编语言语句
- MIPS汇编语言程序模板
- 数据定义
- 内存对齐和字节序
- 系统调用
- 过程
- 参数传递和运行时栈

数据定义

- 为变量的存储划分内存
- 可能会有选择的为数据分配名字（标签）
- 语法:

[名字:] 伪指令 初始值 [, 初始值] . . .

```
var1:  .WORD    10
```

- 所有的初始值在内存中以二进制数据存储

数据伪指令

- .BYTE 伪指令
  - 以8位字节存储数值表
- .HALF 伪指令
  - 以16位（半字长）存储数值表
- .WORD 伪指令
  - 以32位（一个字长）存储数值表
- .WORD w:n 伪指令
  - 将32位数值 *w* 存入 *n* 个边界对齐的连续的字中
- .FLOAT 伪指令
  - 以单精度浮点数存储数值表
- .DOUBLE 伪指令
  - 以双精度浮点数存储数值表

字符串伪指令

- .ASCII 伪指令
  - 为一个ASCII字符串分配字节序列
- .ASCIIZ 伪指令
  - 与.ASCII 伪指令类似, 但是在字符串末尾增加 NULL字符
  - 字符串以NULL结尾，类似C语言
- .SPACE n 伪指令
  - 为数据段中 *n* 个未初始化的字节分配空间
- 字符串中的特殊字符（按照 C 语言的约定）
  - 新行: \n　　Tab:\t　　引用: \"

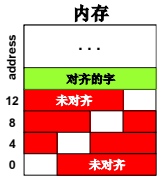
数据定义的例子

```
.DATA
var1:   .BYTE     'A', 'E', 127, -1, '\n'
var2:   .HALF     -10, 0xffff
var3:   .WORD     0x12345678
Var4:   .WORD     0:10
var5:   .FLOAT    12.3, -0.1
var6:   .DOUBLE   1.5e-10
str1:   .ASCII    "A String\n"
str2:   .ASCIIZ   "NULL Terminated String"
array:  .SPACE    100
```

如果初始值超过了值域上界，汇编程序会报错

内存对齐

- 内存可以被看成是带地址的字节数组
  - 字节编址: 地址指向内存中的一个字节
- 字占据内存中4个连续的字节
  - MIPS 指令和整数占据4个字节
- 对齐: 地址是空间大小的整数倍
  - 字的地址是4的整数倍
    - 地址的2位最低有效位必须是 00
  - 半字的地址是 2 的整数倍
- .ALIGN n 伪指令
  - 对下一个定义的数据做 2^n 字节对齐



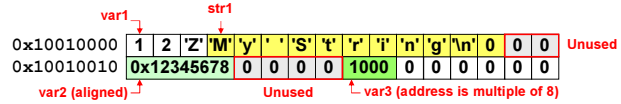
符号表

- 汇编程序为标签（变量）构建符号表
  - 为数据段的每一个标签计算地址
- 例子

```
.DATA
var1:   .BYTE    1, 2,'Z'
str1:   .ASCIIZ  "My String\n"
var2:   .WORD    0x12345678
.ALIGN  3
var3:   .HALF    1000
```

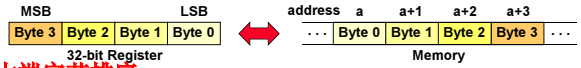
符号表

标签	地址
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018

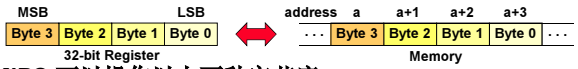


字节序和端

- 处理器对一个字内的字节排序有两种方法
- 小端字节排序
  - 内存地址 = 最低有效字节的地址
  - 例子: Intel IA-32, Alpha



- 大端字节排序
  - 内存地址 = 最高有效字节的地址
  - 例子: SPARC, PA-RISC



- MIPS 可以操作以上两种字节序

- MIPS汇编语言语句
- MIPS汇编语言程序模板
- 数据定义
- 内存对齐和字节序
- 系统调用
- 过程
- 参数传递和运行时栈

系统调用

- 程序通过系统调用实现输入/输出
- MIPS 提供一条特殊的 `syscall` 指令
  - 从操作系统获取服务
- 使用 `syscall` 系统服务
  - 从 $v0寄存器中读取服务数
  - 从 $a0, $a1, 等寄存器中读取参数值（如果有）
  - 发送 `syscall` 指令
  - 从结果寄存器中取回返回值（如果有）

Syscall 服务

Service	$v0	Arguments / Result
Print Integer	1	$a0 = integer value to print
Print Float	2	$f12 = float value to print
Print Double	3	$f12 = double value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	$v0 = integer read
Read Float	6	$f0 = float read
Read Double	7	$f0 = double read
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Exit Program	10	
Print Char	11	$a0 = character to print
Read Char	12	$a0 = character read

读和打印一个整数

```
################## Code segment ######################
.text
.globl main
main:                           # main program entry
  li    $v0, 5                  # Read integer
  syscall                       # $v0 = value read

  move  $a0, $v0                # $a0 = value to print
  li    $v0, 1                  # Print integer
  syscall

  li    $v0, 10                 # Exit program
  syscall
```


读和打印一个串

```
################## Data segment ######################
.data
  str: .space  10           # array of 10 bytes
################## Code segment ######################
.text
.globl main
main:                       # main program entry
  la    $a0, str            # $a0 = address of str
  li    $a1, 10             # $a1 = max string length
  li    $v0, 8              # read string
  syscall
  li    $v0, 4              # Print string str
  syscall
  li    $v0, 10             # Exit program
  syscall
```


例程：3整数求和

```
# Sum of three integers
#
# Objective: Computes the sum of three integers.
#     Input: Requests three numbers.
#    Output: Outputs the sum.
################## Data segment ##################
.data
prompt:   .asciiz     "Please enter three numbers: \n"
sum_msg:  .asciiz     "The sum is: "
################## Code segment ##################
.text
.globl main
main:
      la    $a0,prompt          # display prompt string
      li    $v0,4
      syscall
      li    $v0,5               # read 1st integer into $t0
      syscall
      move  $t0,$v0
```


例程：3整数求和（2）

```
      li    $v0,5               # read 2nd integer into $t1
      syscall
      move  $t1,$v0

      li    $v0,5               # read 3rd integer into $t2
      syscall
      move  $t2,$v0

      addu  $t0,$t0,$t1         # accumulate the sum
      addu  $t0,$t0,$t2

      la    $a0,sum_msg         # write sum message
      li    $v0,4
      syscall

      move  $a0,$t0             # output sum
      li    $v0,1
      syscall

      li    $v0,10              # exit
      syscall
```

例程：大小写转换

```
# Objective: Convert lowercase letters to uppercase
#     Input: Requests a character string from the user.
#    Output: Prints the input string in uppercase.
################### Data segment #####################
.data
name_prompt: .asciiz          "Please type your name: "
out_msg:     .asciiz          "Your name in capitals is: "
in_name:     .space 31        # space for input string
################### Code segment #####################
.text
.globl main
main:
      la    $a0,name_prompt   # print prompt string
      li    $v0,4
      syscall
      la    $a0,in_name       # read the input string
      li    $a1,31            # at most 30 chars + 1 null char
      li    $v0,8
      syscall
```


例程：大小写转换（2）

```
      la    $a0,out_msg       # write output message
      li    $v0,4
      syscall
      la    $t0,in_name
loop:
      lb    $t1,($t0)
      beqz  $t1,exit_loop     # if NULL, we are done
      blt   $t1,'a',no_change
      bgt   $t1,'z',no_change
      addiu $t1,$t1,-32       # convert to uppercase: 'A'-'a'=-32
      sb    $t1,($t0)
no_change:
      addiu $t0,$t0,1         # increment pointer
      j     loop
exit_loop:
      la    $a0,in_name       # output converted string
      li    $v0,4
      syscall
      li    $v0,10            # exit
      syscall
```


- MIPS汇编语言语句
- MIPS汇编语言程序模板
- 数据定义
- 内存对齐和字节序
- 系统调用
- 过程
- 参数传递和运行时栈

过程

- 观察 swap 过程 (C程序)
- 翻译成 MIPS 汇编语言

```
void swap(int v[], int k)
{   int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
 sll $t0,$a1,2    # $t0=k*4
 add $t0,$t0,$a0  # $t0=v+k*4
 lw  $t1,0($t0)   # $t1=v[k]
 lw  $t2,4($t0)   # $t2=v[k+1]
 sw  $t2,0($t0)   # v[k]=$t2
 sw  $t1,4($t0)   # v[k+1]=$t1
 jr  $ra           # return
```

参数:

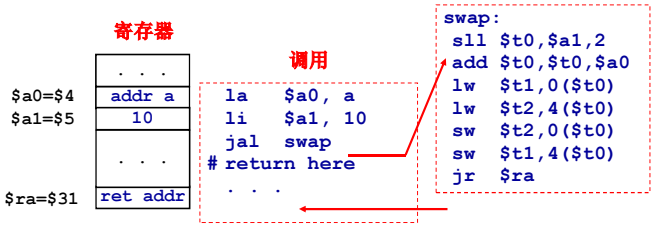
$a0 = v[]的地址

$a1 = k,

返回地址在 $ra 中

调用 / 返回 序列

- 调用swap过程: swap(a,10)
  - 将数组a 的地址和10作为参数传递
  - 调用swap过程，保存返回地址 $31 = $ra
  - 执行swap过程
  - 返回对返回地址的控制



JAL 和 JR的细节

地址	指令	汇编语言	
00400020	lui $1, 0x1001	la $a0, a	伪直接寻址
00400024	ori $4, $1, 0		PC = imm26<<2
00400028	ori $5, $0, 10	li $a1,10	0x10000f << 2
0040002C	jal 0x10000f	jal swap	= 0x0040003C
00400030	. . .	# return here	
			$31 0x00400030
		swap:	
0040003C	sll $8, $5, 2	sll $t0,$a1,2	
00400040	add $8, $8, $4	add $t0,$t0,$a0	
00400044	lw $9, 0($8)	lw $t1,0($t0)	
00400048	lw $10,4($8)	lw $t2,4($t0)	
0040004C	sw $10,0($8)	sw $t2,0($t0)	
00400050	sw $9, 4($8)	sw $t1,4($t0)	
00400054	jr $31	jr $ra	

寄存器 $31 是返回地址寄存器

过程的指令

- JAL (Jump-and-Link) ：调用指令
  - 在 $ra = PC+4 中保存返回地址并跳转到相应的过程
  - 寄存器 $ra = $31 被 JAL 用来保存返回地址
- JR (Jump Register) ： 返回指令
  - 跳转到在 寄存器Rs (PC = Rs)中存储的地址所在指令
- JALR (Jump-and-Link Register)
  - 在Rd = PC+4中存储返回地址,
  - 跳转到在寄存器Rs (PC = Rs)中存储的地址所在过程
  - 用于调用方法(地址仅在运行时可知)

Instruction		Meaning	Format					
jal	label	$31=PC+4, jump	$op^6 = 3$	imm^{26}				
jr	Rs	PC = Rs	$op^6 = 0$	rs^5	0	0	0	8
jalr	Rd, Rs	Rd=PC+4, PC=Rs	$op^6 = 0$	rs^5	0	rd^5	0	9

- MIPS汇编语言语句
- MIPS汇编语言程序模板
- 数据定义
- 内存对齐和字节序
- 系统调用
- 过程
- 参数传递和运行时栈

参数传递

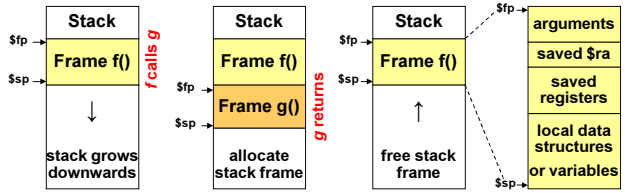
- 汇编语言中的参数传递比高级语言中复杂
- 汇编语言中
  - 将所有需要的参数放置在一个可访问的存储区域
  - 然后调用过程
- 会用到两种类型的存储区域
  - 寄存器: 使用通用寄存器 (寄存器方法)
  - 内存: 使用栈 (栈方法)
- 参数传递的两种常用机制
  - 值传递: 传递参数值
  - 引用传递: 传递参数的地址

参数传递（续）

- 按照约定, 参数传递通过寄存器实现
  - `$a0 = $4 .. $a3 = $7` 用来做参数传递
  - `$v0 = $2 .. $v1 = $3` 用来表示结果数据
- 其它的参数/结果可以放在栈中
- 运行时栈用于
  - 不适合使用寄存器时用来存储变量/数据结构
  - 过程调用中保存和恢复寄存器
  - 实现递归
- 运行时栈通过软件规范实现
  - 栈指针 `$sp = $29` (指向栈顶)
  - 帧指针 `$fp = $30` (指向过程帧)

栈帧

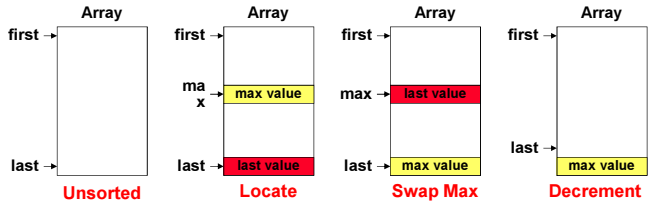
- 栈帧是栈的一段，包含 …
  - 保存的参数，寄存器和本地数据结构
- 也称为活动帧或活动记录
- 帧通过调整指针压入和弹出
  - 栈指针 `$sp = $29` 和帧指针 `$fp = $30`
  - 减指针 `$sp` 分配堆栈帧, 加指针释放



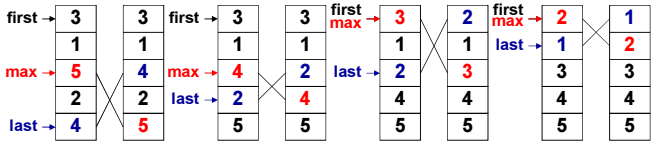
保存寄存器

- 过程调用时需要保存寄存器
  - 栈可用来保存寄存器值
- 什么寄存器需要保存？
  - 在被调过程修改，而调用过程还会再用到的寄存器
- 谁来保存寄存器值？
  - 被调过程: 模块化代码的首选方法
    - 寄存器保存在被调过程内部实现
  - 通常约定, 寄存器 $s0, $s1, …, $s7 需要被保存
  - 同样, 寄存器 $sp, $fp, 和 $ra 需要被保存

选择排序



❖例子



选择排序过程

```
# Objective: Sort array using selection sort algorithm
#     Input: $a0 = pointer to first, $a1 = pointer to last
#    Output: array is sorted in place
#############################################################
sort: addiu $sp, $sp, -4     # allocate one word on stack
      sw    $ra, 0($sp)      # save return address on stack
top:  jal   max              # call max procedure
      lw    $t0, 0($a1)      # $t0 = last value
      sw    $t0, 0($v0)      # swap last and max values
      sw    $v1, 0($a1)
      addiu $a1, $a1, -4     # decrement pointer to last
      bne   $a0, $a1, top    # more elements to sort
      lw    $ra, 0($sp)      # pop return address
      addiu $sp, $sp, 4
      jr    $ra              # return to caller
```


找最大值过程

```
# Objective: Find the address and value of maximum element
#     Input: $a0 = pointer to first, $a1 = pointer to last
#    Output: $v0 = pointer to max,   $v1 = value of max
#############################################################
max:  move  $v0, $a0         # max pointer = first pointer
      lw    $v1, 0($v0)      # $v1 = first value
      beq   $a0, $a1, ret    # if (first == last) return
      move  $t0, $a0         # $t0 = array pointer
loop: addi  $t0, $t0, 4      # point to next array element
      lw    $t1, 0($t0)      # $t1 = value of A[i]
      ble   $t1, $v1, skip   # if (A[i] <= max) then skip
      move  $v0, $t0         # found new maximum
      move  $v1, $t1
skip: bne   $t0, $a1, loop   # loop back if more elements
ret:  jr    $ra
```


递归过程示例

int fact(int n) { if (n<2) return 1; else return (n*fact(n-1)); }

```
fact:  slti   $t0,$a0,2     # (n<2)?
       beq    $t0,$0,else   # if false branch to else
       li     $v0,1         # $v0 = 1
       jr     $ra           # return to caller
else:  addiu  $sp,$sp,-8    # allocate 2 words on stack
       sw     $a0,4($sp)    # save argument n
       sw     $ra,0($sp)    # save return address
       addiu  $a0,$a0,-1    # argument = n-1
       jal    fact          # call fact(n-1)
       lw     $a0,4($sp)    # restore argument
       lw     $ra,0($sp)    # restore return address
       mul    $v0,$a0,$v0   # $v0 = n*fact(n-1)
       addi   $sp,$sp,8     # free stack frame
       jr     $ra           # return to caller
```